# WEST

Help | Logout | Interrupt

Main Menu | Search Form | Posting Counts | Show S Numbers | Edit S Numbers | Preferences | Cases

## Search Results -

| Term | Documents |
|---|---|
| "6298370".USPT. | 1 |
| 6298370S | 0 |
| "6298370".PN..USPT. | 1 |
| (6298370.PN.).USPT. | 1 |

**Database:**
- US Patents Full-Text Database
- US Pre-Grant Publication Full-Text Database
- JPO Abstracts Database
- EPO Abstracts Database
- Derwent World Patents Index
- IBM Technical Disclosure Bulletins

**Search:** L43

Refine Search

Recall Text | Clear

## Search History

**DATE: Monday, May 19, 2003** Printable Copy Create Case

| Set Name side by side | Query | Hit Count | Set Name result set |
|---|---|---|---|
| | *DB=USPT; PLUR=YES; OP=ADJ* | | |
| L43 | 6298370.pn. | 1 | L43 |
| L42 | 5590334.pn. | 1 | L42 |
| L41 | 4658351.pn. | 1 | L41 |
| L40 | 5388219.pn. | 1 | L40 |
| L39 | L38 and l11 | 3 | L39 |
| L38 | (first or second) near1 l3 | 492 | L38 |
| L37 | L36 and l11 | 3 | L37 |
| L36 | L35 near3 l2 | 44 | L36 |

| | | | |
|---|---|---|---|
| L35 | routine$1 | 142399 | L35 |
| L34 | l11 and l32 | 5 | L34 |
| L33 | L32 and l31 and l11 | 0 | L33 |
| L32 | event queue | 790 | L32 |
| L31 | process manager | 282 | L31 |
| L30 | process manager | 282 | L30 |
| L29 | 2516105.pn. | 1 | L29 |
| L28 | Mateosian | 5 | L28 |
| L27 | Mateosian.in. | 0 | L27 |
| L26 | L25 and l24 | 8 | L26 |
| L25 | L2 and l4 and l5 | 414 | L25 |
| L24 | l3.ab. | 544 | L24 |
| L23 | L22.ab. | 8 | L23 |
| L22 | task control near1 block | 285 | L22 |
| L21 | Matteosian.in. | 0 | L21 |
| L20 | Mateosian,Richard.in. | 0 | L20 |
| L19 | Mateo$.In. | 27 | L19 |
| L18 | Mateosian.In. | 0 | L18 |
| L17 | Mateosian.in. | 0 | L17 |
| L16 | l15 and l2 | 0 | L16 |
| L15 | koning.in. | 187 | L15 |
| L14 | L13 and l12 | 11 | L14 |
| L13 | procedure call | 3813 | L13 |
| L12 | L11 and l8 and l9 | 85 | L12 |
| L11 | l2 and l3 and l4 and l5 | 136 | L11 |
| L10 | l2 and l3 and l4 and l5 and l6 and l7 and l8 and l9 | 0 | L10 |
| L9 | real time$1 | 87727 | L9 |
| L8 | exception$1 | 154785 | L8 |
| L7 | context information$1 | 1219 | L7 |
| L6 | remote procedure near1 call | 1485 | L6 |
| L5 | pointer$1 | 60016 | L5 |
| L4 | memory space | 14626 | L4 |
| L3 | control block$1 | 15229 | L3 |
| L2 | context switch$ | 1986 | L2 |
| L1 | 5918021.pn. | 1 | L1 |

END OF SEARCH HISTORY

WEST

☐ Generate Collection Print

L26: Entry 6 of 8 File: USPT Dec 31, 1996

DOCUMENT-IDENTIFIER: US 5590334 A
TITLE: Object oriented message passing system and method

Abstract Text (1):
An object oriented message passing system for transferring messages between a client task and a server task comprises an object database, an object management unit, a message transaction unit, and a locking unit. The object management unit creates a port object and one or more associated message objects. The message transaction unit matches a send message request issued by a client task with an acceptance function or with a receive message request issued by a server task. In response to a send message request, the message transaction unit creates a send message control block. In response to a receive message request, the message transaction unit creates a delivery message control block if the receive message request matches the send message control block, or creates a receive message control block if the receive message request does not match the send message control block. The locking unit locks a message object such that send message requests directed to the message object are not eligible to be matched to receive message requests until the message object is unlocked.

Abstract Text (2):
An object oriented message passing method comprises the steps of: creating a port object; creating a message object associated with the port object; generating a unique message ID in response to a message transaction initiated by a send message request; creating a send message control block; and matching the send message control block to a corresponding receive message request.

Brief Summary Text (14):
The object oriented message passing unit associates an acceptance function with a port object upon request, where the acceptance function provides a means for performing one or more services within the context and address space of the client task. Acceptance functions significantly reduce the amount of time required to complete time-critical services by eliminating the need for mapping between address spaces and context switching.

Detailed Description Text (10):
A seventh data field in the port object 54 optionally specifies an acceptance function. The acceptance function comprises a set of instructions that directly implements a subset of services provided by a server task 34 within the task context of a client task 32. The acceptance function uses the microkernel-accessible address area that is common to the address space of the client task 32, and therefore effectively functions within the address space of the client task 32. Acceptance functions thus eliminate the need for context switching and mapping between address spaces. Performance of a given service via an acceptance function therefore requires much less computational time than performance of the same service via a server task 34. Acceptance functions provide a means for minimizing the amount of time required to perform time-critical operations. The seventh data field in the port object 54 also specifies a set of message types for which the acceptance function is capable of providing a service. Preferably, the seventh data field is empty when the object management unit 42 first creates the port object 54. The object management unit 42 stores or registers a reference to an acceptance function and the set of message types in response to a server task registration request that identifies a particular acceptance function and the set of message types.

Detailed Description Text (11):
In an exemplary situation in which acceptance functions might be used beneficially, disk input/output (I/O) operations may require services provided by a first server task 34 associated with a file system. The first server task 34 may selectively require particular services provided by a second server task 34 associated with a disk driver, which may in turn selectively require particular services provided by a third server task 34 associated with a small computer systems interface (SCSI) manager. If the second server task 34 and the third server task 34 issue appropriate registration requests, the object management unit 42 will register an acceptance function for the second server task 34 and an acceptance function for the third server task 34, respectively. Those disk I/O operations that require the particular services corresponding to the acceptance functions registered will occur within the task context and within the address space of the first server task 34, eliminating the need for mapping between address spaces and context switching. This in turn will greatly reduce the time required to perform these disk I/O operations.

Detailed Description Text (15):
In the preferred embodiment, the object management unit 42 can associate multiple message objects 52 with a single port object 54. The memory storage requirements for each message object 52 are significantly less than the memory storage requirements for each port object 54. In an exemplary embodiment, each port object 54 occupies 128 bytes within the memory 20, while each message object occupies as little as 24 bytes. In the present invention, because a given server task 34 can register an acceptance function that is to provide one or more services, the given server task 34 is simpler and requires less memory to implement. Moreover, because an acceptance function executes within the task context of a client task 32, no additional memory is required for context switching when an acceptance function performs a service. Thus, the object oriented message passing model 50 provided by the present invention requires significantly less memory space than that required by any message passing model provided by prior art message passing systems, while providing a higher level of structural granularity for a given amount of available memory.

Detailed Description Text (58):
Next, in step 216, the message transaction unit 44 determines whether an acceptance function specifying a message type that matches the message type in the send MCB has been registered with the port object 54. If an acceptance function has been registered, the message transaction unit 44 creates a delivery MCB 80 in the client task's address space in step 218. Preferably, the delivery MCB 80 is created in the microkernel-accessible portion of the client task's address space. Following step 218, the message transaction unit 44 passes a pointer to the delivery MCB 80 to the acceptance function in step 220. Next, the message transaction unit 44 inserts a reference to the send MCB in the pending reply message list within the port object 54 in step 222.

**WEST**

**End of Result Set**

☐ Generate Collection Print

L37: Entry 3 of 3 File: USPT Feb 7, 1995

DOCUMENT-IDENTIFIER: US 5388219 A
TITLE: Efficient channel and control unit for host computer

Abstract Text (1):
An I/O system including a processor, a multitasking operating system and DMA hardware efficiently controls a transfer of data between a main memory and memories of different types of devices by minimizing context switches between tasks and wait times of the tasks. A plurality of validation routines are used to validate a plurality of commands when the validation routines are called. Each of the commands corresponds to a specific type of I/O operation and a specific one of the device memories to participate in the I/O operation with the main memory. Each of the validation routines is device type specific and command type specific. A general routine responds to each of the commands by identifying and calling the validation routine which corresponds to the type of I/O operation and type of device which are specified in the command. The general routine initiates I/O hardware after the validation routine validates the command. After the I/O hardware completes the I/O operation, it signals a command completion routine which is command specific and device type specific. In response, the command completion routine signals to the general routine a state of the I/O operation. Each of the validation routines executes on the same task as the general routine to minimize context switches, and each of the command completion routines executes on a different task than the general routine to minimize wait time for the command completion routine.

Brief Summary Text (5):
A later version of this System/370 computer system substitutes a single program which directly emulates the function of the Block Multiplexor channel and the function of each control unit. This program is executed by a multitasking operating system and provides one task to emulate the Block Multiplexor channel and another task to emulate each of the control units. Because these tasks directly emulate the Block Multiplexor channel and associated control units there is much interplay that requires many "context switches" between the channel task and each control unit task during the course of implementing a single read or write operation. A "context switch" occurs when one task stops executing on a processor and another task begins to execute on a processor. For each context switch, there is considerable overhead expended in recording where the one task stops executing and recording the contents of the associated registers and loading the registers for the other task. In the case of a write CCW within this computer system, context switches are required when the channel passes the CCW command to the control unit, the control unit passes initial status back to the channel, the channel signals an acceptance of the initial status to the control unit, the control unit requests data from the channel, and the control unit presents ending status to the channel. Also, considerable processor time is wasted by the channel task in waiting for the write or read operation to complete.

Brief Summary Text (8):
The invention resides in an I/O system which efficiently controls a transfer of data between a main memory and memories of different types of devices. The I/O system comprises a processor, a multitasking operating system for controlling program execution on the processor, and a channel and control unit program which executes on the processor. The program is divided into the following program routines which execute on the same or separate tasks as indicated, to minimize the number of

context switches required to execute each command and the delay to each task.

Brief Summary Text (11):
After the I/O hardware completes the I/O operation, it signals a command completion routine which is device type specific. In response, the command completion routine signals to the general routine a state of the I/O operation. Each of the validation routines executes on the same task as the general routine. The command completion routine executes on a different task than the general routine. Consequently, no context switch is required between the general routine and the validation routine. This optimizes the speed at which the command is executed. While a context switch is required from the general routine to the command completion routine, this context switch is overlapped with the DMA operation so time is not wasted. A context switch is also required from the command completion routine to the general routine after the DMA operation. Neither the command completion routine nor the general routine wait while the data movement is performed by the I/O hardware. This optimizes the use of the I/O processor.

Detailed Description Text (6):
After building the channel program, the operating system 13 calls the channel routine 20 by making a "Start Subchannel" instruction (step 110 of FIG. 2). The channel routine accesses a control block 112 in RAM 51 corresponding to the device being accessed, which control block includes pointers to the CCW table 28, 29 or 33 and routine address table 38, 39 or 43 corresponding to the type of device, and a pointer to a data buffer 114 and a control block 113 in the control unit memory corresponding to the device. The information in the instruction and the channel program is used to update the control block with the type of command, byte length and data address information and other parameters in the CCW. The control block also stores the state of any operation in progress for the device and the control unit routine, which states were previously entered by the channel routine. As described in more detail below, the control unit comprises the CCW validation routines 21 and command completion routines 22.

Detailed Description Text (8):
FIG. 3 illustrates the CCW tables 28, 29 and 33 and routine address tables 38, 39 and 43 in more detail. The address of one of each is stored in the control block 112 corresponding to the device to be accessed. The channel routine uses the command-type information from the CCW to identify a corresponding command specific entry in the CCW table. The command specific entry includes a byte count which indicates the length of data that the channel routine needs to prefetch for the specific device-type and command-type, and an index into the routine address table 38, 39 or 43.

Detailed Description Text (9):
Assuming that the byte count field is greater than zero, the channel routine prefetches data from the main memory location indicated by the data address field of the CCW which is stored in the control block 112, and stores the data in buffer 114 (step 140). The foregoing operation is considered a "prefetch" because the data will be subsequently used by a CCW validation routine of the control unit which is not yet executing and has not yet asked for the data. The channel routine uses the other field of the CCW table command specific entry as an index to identify a command specific entry from the corresponding routine address table (step 150).

Detailed Description Text (10):
The identified entry in the routine address table points to a location in RAM 51 which begins a CCW validation routine. FIG. 4 illustrates a multiplicity of CCW validation routines 21. Each CCW validation routine is subchannel/device specific and command specific. The channel routine then calls that CCW validation routine and passes the address of the appropriate subchannel control block 112 as a parameter.

Detailed Description Text (11):
The CCW validation routine is a control unit routine but executes on the common task to avoid an inefficient context switch. In step 160, the CCW validation routine determines if the command is a type which the control unit can handle, and whether the control unit is in a proper state (i.e. available) to process the CCW. The following are examples of types of CCWs that a control unit for a fixed block DASD

type of device is capable of processing:

Detailed Description Text (42):
CWRITE--control block write

Detailed Description Text (44):
CREAD--control block read

Detailed Description Text (48):
Both the control unit and the device must be available for the CCW validation routine to accept the CCW. Usually both the control unit and the device exhibit the same state. However, if the control unit services more than one device (of the same type), then the control unit can be busy when one of the devices is available. Also, some devices can perform work without assistance from the respective control unit, in which case, the device can be busy when the control unit is available. If either the control unit or device is busy, then the channel routine will redrive the subchannel later and the foregoing steps will be repeated. The CCW validation routine continues with step 170 when the CCW is an acceptable type and the control unit and device are both available. In step 170, the CCW validation routine changes the control unit state and the corresponding device state to "busy with this channel program" by storing such status information in the respective control block 113. Then, using the parameters passed by the channel routine and the control data which was prefetched into the respective data buffer 114, the CCW validation routine is able to execute the control command. This assumes that all of the control CCW control data was prefetched, which is normally the case because most control CCWs have a small byte count. Next, the CCW validation routine returns to the channel routine and passes unit status as parameters (step 180). The "unit" status indicates whether the CCW validation routine accepted the CCW and, if so, the action that the CCW validation routine took in response to the CCW. Zero initial status indicates that the command was accepted, and primary status of channel end and device end indicates that the command has been executed without error. The other statuses are described in the Principles of Operation publication previously referenced.

Detailed Description Text (49):
Next, the channel routine signals operating system 13 that it has begun executing the channel program (step 190). At this point, the first (control) CCW of the channel program has been completely executed within the context of the common task with no task or context switches.

Detailed Description Text (50):
Based on the content of the flag byte from the first CCW, the channel routine which is still executing on the common task, continues execution of the channel program (step 190). The channel routine reads the next (write) CCW of the channel program from a location in main memory 42 sequential to the previous CCW and stored in control block 112 (step 200). The write CCW includes command, data address, byte count and flag fields. The byte count field indicates the length of data that the application program stored in main memory 42 to be written to the device pursuant to this CCW, and the data address field indicates the location of this data in main memory 42. (Operating system 13 can divide data that the application program desires to write between two or more write CCWs.) The flag field indicates whether there are additional CCWs in the channel program. The channel routine continues to access the same subchannel control block 112 which identifies the corresponding CCW table and routine address table. The command field in the CCW identifies a corresponding command specific entry from the CCW table.

Detailed Description Text (51):
The command specific entry includes a byte count which indicates the length of data that the channel routine needs to prefetch for this specific device type and command-type, and an index into the routine address table. If the byte count field is greater than zero, the channel routine prefetches the data from the main memory location indicated by the data address field of the CCW, and stores the data in the respective data buffer 114 in RAM 51 which is accessible by the control routine (step 210). For this example, assume that the data block in main memory 42, which is addressed by the CCW data address, consists of two portions, a relatively small amount of control data, and a larger block of data to be written to the device. The

control data is a header which indicates the destination (file, block, track or record) of the remaining data within the device. Only the control data has been prefetched. The channel routine uses the other field of the CCW table entry as an index to identify a command specific entry from the respective routine address table (step 220). The respective routine address table entry contains the location in RAM 51 of the CCW validation routine that corresponds to this device type and the write CCW. The channel routine then calls that CCW validation routine and passes the CCW address of the appropriate subchannel control block 112 as a parameter (step 222).

Detailed Description Text (52):
The CCW validation routine is considered part of the control unit but executes on the common task to avoid an inefficient context switch. In step 230, the CCW validation routine determines if it should accept the command based on whether this CCW is a type which the control unit is capable of processing and whether the control unit and device are both available. Devices 18, 19 and 23 (as well as most devices) are capable or processing a write CCW. The control unit and device are both available if both are not busy at all or if both are busy with this channel program. Because this is a continuation of a channel program which is already in progress, both the control unit and device will be available. The CCW validation routine continues with step 240 if the CCW is an acceptable type. In step 240, the CCW validation routine compares the byte count parameter passed by the channel routine to the byte count stored in the CCW table, and determines if there is additional, device data to be fetched from the main memory. In the illustrated example this is the case, so the CCW validation routine returns an initial status of zero (indicating that the write command is accepted), requests more data from the channel routine and provides the address of another buffer memory in device 18, 19 or 23 to receive the device data (step 250).

Detailed Description Text (54):
Next, the command completion routine stores primary status of channel end and device end in the respective control block 112 to indicate to the channel routine (when subsequently executed) that the write operation has been successfully completed (step 290). Next, the command completion routine posts the common task and exits the device-type specific task (step 292).

Detailed Description Text (55):
The channel routine resumes execution on the common task and reads the next (read) CCW of the channel program from RAM 42 (step 310). The read CCW includes command, data address, byte count and flag fields. The byte count field indicates the amount of data that the application program desires to read from the device, and the data address field indicates the location in main memory 42 to store the data which will be read. The flag field indicates that there are no more CCWs in this channel program. The channel routine continues to access the same subchannel control block 112 which identifies the corresponding CCW table and routine address table. The channel routine uses the command type in the read CCW to identify a corresponding command specific entry from the CCW table.

Detailed Description Text (56):
The command specific entry includes a byte count which should be zero because this is a read operation and there is no control or actual data to prefetch. The command specific entry also includes an index into the corresponding routine address table. Because the byte count equals zero, the channel routine does not attempt to prefetch control or actual data from the main memory. The index into the routine address table identifies a CCW validation routine that is specific to the type of device to be accessed and the type of command. The channel then calls that CCW validation routine and passes the address of the appropriate subchannel control block 112 as a parameter (step 320).

Detailed Description Text (57):
The CCW validation routine is considered part of the control unit but executes on the common task to avoid an inefficient context switch. In step 330, the CCW validation routine determines if the control unit should accept the command based on whether the CCW is a type which the control unit and device are capable of processing and whether the control unit and device are both available for this channel program. Devices 18, 19 and 23 (as well as most devices) are capable of

processing a read CCW. Since this is a continuation of a channel program which is already in progress, both the control unit and device will be available. If the CCW is accepted, the CCW validation routine continues with step 340 in which the CCW validation routine determines from the nature of the command that data must be transferred from the device to main memory 42, and therefore, returns a request to move data to the specified location in the main memory.

Detailed Description Text (59):
After the data transfer from the device to the main memory is completed, the Blue bus interface controller 40 interrupts the I/O processor to resume execution of the command completion routine within the device-type specific task. A DMA end interrupt handler routine which handles the interrupt knows which device-type specific task to post based on information stored in RAM 51 by the channel routine when the DMA was initiated. In response, the command completion routine completes execution of the read command by verifying that all of the data was moved without error (step 360). Then the command completion routine stores primary status of channel end and device end in the respective control block 112 to indicate to the channel routine (which will subsequently execute) that the read operation has been successfully completed (step 370). Next, the command completion routine posts the common task, and exits the device-type specific task (step 372).

Detailed Description Text (60):
Now, the channel routine resumes execution, and the channel routine recognizes from the control block 112 that the read CCW has successfully completed. The channel routine also recognizes from flag field information stored in control block 112 that the end of the channel program has been reached (step 380). Next, the channel routine reads another entry from the routine address table which entry is located at a fixed offset into the routine address table and indicates the location of a control unit specific "operation complete" routine (step 390). The channel routine calls this routine in step 392. The operation complete routine executes in the context of the common task to provide end of operation processing. During such processing, the operation complete routine recognizes the end of operation (Step 400) and then resets the control block 112 to indicate that the control unit state and device state are now both available (step 410). Then, the operation complete routine returns to the channel routine (step 420).

Detailed Description Text (62):
It should be noted that execution of the entire channel program consisting of the control, write and read CCWs required a context switch in the critical path only after step 290 and step 372. Task switches after step 260 and step 350 do not add to the execution time of the channel program because they are overlapped with the DMA data movement. This greatly improves efficiency compared to the five task switches in the respective critical paths required to execute each CCW in the prior art described above in the Background Of The Invention for the first IBM System/370 computer system which provided an integrated channel and control unit program to replace a physical channel and control unit.

Detailed Description Text (66):
The processor bus interface 48 comprises a local processor bus for the I/O processor 15 and transceivers 450-452. As noted above, the channel for I/O processor 15 routine in step 260 initiates the controllers to perform the write operation. As part of the initiation, the channel routine writes the starting address of the main memory and the byte count into a DMA port control block 502 in the Blue bus interface controller. Also, the channel routine writes the starting address of the memory of the device to receive the data as well as the byte count into a DMA control register 504 within the Micro Channel (R) interface controller. Next, the channel routine writes a command into the DMA control register 504 to move the data from a bypass bus 506 to the Micro Channel (R) bus 46. Then, the channel routine writes a command into the DMA port control block 502 to move the data from the Blue bus 41 to the bypass bus.

Detailed Description Text (67):
Host interface control logic 510 reads the command, address and byte count from the DMA port control block 502, fetches the data from the main memory and stores the data into two DMA port data buffers 512 in the Blue bus interface controller. As the

host interface control logic 510 fetches and stores the data, the host interface control logic also increments the host memory address and decrements the byte count in the DMA port control block. When one of the DMA port data buffers becomes full, the host interface control logic checks whether the other DMA port data buffer is empty. If so, the host interface control logic fills this other DMA port data buffer. When both DMA port data buffers are full, the host interface control logic waits for one to become empty as follows.

Detailed Description Text (69):
When one of the DMA data buffers becomes full, Micro Channel (R) interface control logic 524 reads the device memory starting address and byte count from the DMA control registers 504, fetches the data from the DMA data buffers and stores the data in the memory of the target device 18 or 19. While, the control logic 524 fetches and stores the data, the control logic 524 also increments the device memory address and decrements the byte count in the DMA control registers. When the byte count in the DMA port control block reaches zero, the control logic 510 generates an interruption request to the I/O subsystem processor 15. The interruption of the I/O subsystem processor cause an interrupt handler to run. The interrupt handler reads status information and residual byte count from the DMA port control block 502 within the Blue bus interface controller, and status information and residual byte count from the DMA control registers 504 within the Micro Channel (R) interface controller to determine if the write operation completed successfully. The interrupt handler then posts the appropriate device-type specific task to execute in step 280 and also indicates that the write operation completed successfully. If the write operation did not complete successfully, the interrupt handler passes the status and byte count information to the device-type specific task.

Detailed Description Text (70):
In step 350 of FIG. 2, the channel routine initiates a read operation. As part of the initiation, the channel routine writes the starting address of the main memory to receive the data and the byte count into the DMA port control block 502 within the Blue bus interface controller. Also, the channel routine writes the starting address of the device memory from which the data should be read and the byte count into the DMA control register 504 in the Micro Channel (R) interface controller. The channel routine also writes a command into the DMA control register 504 to move data from the Micro Channel (R) bus to the bypass bus, and a command into the DMA port control block 502 to move data from the bypass bus to the Blue bus.

Detailed Description Text (73):
When one of the DMA port data buffers 512 becomes full, the host interface control logic 510 fetches this data and stores this data in main memory 42 at an address stored in the DMA port control block 502. As the control logic fetches and stores this data, the control logic also increments the main memory address and decrements the byte count in the DMA port control block. When the byte count in the DMA port control block becomes zero, the control logic 510 generates an interruption request to the I/O subsystem processor. The interruption to the I/O subsystem processor causes an interrupt handler to begin executing. The interrupt handler reads status information and the residual byte count in the DMA port control block 502 within the Blue bus interface controller, and status information and the residual byte count in the DMA control register 504 within the Micro Channel (R) interface controller to determine if the read operation completed successfully. Then the interrupt handler posts the appropriate device-type specific task and indicates that the read operation has completed successfully. The operation complete routine begins to execute in step 360 of FIG. 2. If the read operation did not complete successfully, the interrupt handler passes the status and byte count information to the device-type specific task.

Other Reference Publication (2):
IBM Technical Disclosure Bulletin, vol. 31, No. 4, Sep. 1988, "Implementing Code to Run in Paged Memory Space", pp. 110-111.

Other Reference Publication (3):
Computer Journal, vol. 28, No. 4, Aug. 1985, London, Great Britian, "On Efficient Context Switching", pp.375-378, A. Ramsay.

WEST

☐ Generate Collection Print

L14: Entry 1 of 11 File: USPT Oct 2, 2001

DOCUMENT-IDENTIFIER: US 6298370 B1
TITLE: Computer operating process allocating tasks between first and second processors at run time based upon current processor load

Drawing Description Text (20):
FIG. 18 is a process diagram or method-of-operation diagram showing interrelated improved processes called DirectDSP, DirectDSP HEL (host emulation), DirectDSP HAL (hardware abstraction layer), and VSP Kernel (DSP Real-Time Kernel) herein;

Drawing Description Text (32):
FIG. 28 is a diagram of memory spaces representing a shared memory model utilized in embodiments of processes, devices and systems herein;

Drawing Description Text (34):
FIG. 29 is a diagram of interrupt levels utilized in connection with hardware interrupts and deferred procedure calls (DPCs) in process, device and system embodiments;

Drawing Description Text (35):
FIG. 30 is a further diagram of interrupt levels over time utilized in connection with hardware interrupts and deferred procedure calls (DPCs) in process, device and system embodiments;

Drawing Description Text (36):
FIG. 31 is a classification diagram of interrupt levels in real-time and dynamic classes in connection with process, device and system embodiments;

Drawing Description Text (41):
FIG. 36 is a memory space diagram of host memory program and data spaces (at left) and DSP on-chip and off-chip memories (at right) representing an example of a shared memory model utilized in embodiments of processes, devices and systems herein;

Drawing Description Text (43):
FIG. 38 is a DSP memory space diagram supplementing FIG. 36-right and showing DSP program, data and I/O spaces, including on-chip and off-chip memories and registers utilized in embodiments of processes, devices and systems herein;

Drawing Description Text (44):
FIG. 39 is a memory space diagram of host memory program and data spaces (at top) and DSP memory space (at bottom) representing an example of handles and data structures in the shared memory model of FIG. 36 utilized in FIG. 33 sound-related embodiments of processes, devices and systems herein;

Drawing Description Text (46):
FIG. 41 is a memory space diagram showing improved coupling between Host spaces, PCI spaces, and DSP spaces in system embodiments.

Drawing Description Text (48):
FIG. 43 is a real-time-flow diagram of four processes (PCI Bus Master ISR, DSP Message Handler, Audio Out Task, Mixer ISR) in the audio process of FIG. 33 in an example of single-tasking VSP kernel execution;

Drawing Description Text (49):
FIG. 44 is a flow chart diagram of an example of message processing, combined with a memory space diagram of host memory (at top) and DSP memory (at bottom) representing an example of handles, objects and data structures in the shared memory model of FIG. 36 utilized in FIG. 33 wave-sound and other embodiments of processes, devices and systems herein;

Drawing Description Text (63):
FIG. 54A is a partially-schematic, partially real-time process flow diagram of an eight-byte read with byte alignment in an example using 3 PCI data phases in the process of FIG. 54;

Drawing Description Text (64):
FIG. 54B is a partially-schematic, partially real-time process flow diagram of a nine-byte read with byte alignment in an example using 3 PCI data phases in the process of FIG. 54;

Drawing Description Text (65):
FIG. 54C is a partially-schematic, partially real-time process flow diagram of a five-byte read with byte alignment and byte padding in an example using 2 PCI data phases in the process of FIG. 54;

Drawing Description Text (71):
FIG. 54I is an electrical block diagram of PCI host accessible registers starting at base address BA0 in PCI I/O space and replicated and starting at base address BA1 in PCI memory space of FIG. 128, (BA0, BA1 defined in PCI configuration register 0x10, 0x14), and FIG. 54I further indicates address offset decodes and read or read/write circuits associated with those PCI host accessible registers in the wrapper ASIC of VSP;

Drawing Description Text (80):
FIG. 57A is a memory space diagram of host main DRAM memory showing memory allocation and pages locked during initialization in a shared memory model method and system embodiment;

Drawing Description Text (81):
FIG. 57B is a memory space diagram of host main DRAM memory showing memory allocation and pages scatter-locked in a shared memory model method and system embodiment for source/destination data DMA transfers;

Drawing Description Text (82):
FIG. 57C is a memory space diagram of host main DRAM memory showing memory allocation and regions locked in a shared memory model method and system embodiment for source DMA transfer table;

Drawing Description Text (83):
FIG. 57D is a memory space diagram of host main DRAM memory showing a page list structure in a shared memory model method and system embodiment for stream I/O processing;

Drawing Description Text (84):
FIG. 57E is a memory space diagram of host main DRAM memory showing memory allocation and regions locked in a shared memory model method and system embodiment for destination DMA transfer table;

Drawing Description Text (85):
FIG. 57F is a memory space diagram of host main DRAM memory howing a DSP message queue and a host message queue with host manipulated head and tail pointers on the left side, and DSP manipulated head and tail pointers on the right side;

Drawing Description Text (96):
FIG. 67 is a diagram of wrapper ASIC DPRAM memory space for DSP bootload purposes, the memory space pointed to by an SRC address of FIG. 70;

Drawing Description Text (114):

FIG. 84 is an electrical block diagram of components and architecture of an improved real-time private bus-connected VSP-graphics/video chip and VSP-comm-audio-cardbus chip in a system embodiment improved by unified signal processing herein;

Drawing Description Text (115):
FIG. 85 is an electrical block diagram of components and architecture of an improved real-time private bus-connected graphics/video chip and VSP-comm-audio-cardbus in a further improved multimedia system embodiment improved by unified signal processing herein;

Detailed Description Text (6):
Down scaled performance is an annoyance in recalculating a spreadsheet. But for decoding a movie, using Internet telephony, or tele-gaming, downward scaling means losing real world data and compromising quality of service and accuracy. When real-time media streaming functions lack enough MIPS to run, catastrophic failure results.

Detailed Description Text (7):
A statically balanced system does not prevent non-scalable real time functions from failing and scalable operations do not scale upward even though unused MIPS exist in the system.

Detailed Description Text (17):
A multimedia extension MMX single instruction multiple data (SIMD) unit inside the CPU can accelerate host emulation of some of the more real-time applications such as video and to some extent parallel pixel operations, using x86 emulation code ported to MMX code. However, issues include inefficient physical partitioning, integration, and concurrency of highly specialized processing elements. Since MMX is on-host and on-chip it competes directly with other x86 processing units for system resources.

Detailed Description Text (19):
In some improved system embodiments, the OS controls multiple modular, stackable, concurrent computational resources (processors or hardware accelerators), and the improved system supports a wider variety of multimedia device emulation tasks. Modularity adds processing MIPs or elements, and the improved system gracefully orchestrates their operation with the host CPU/MMX for audio, video, graphics, communication and other functions. These modular and distributed processing elements in the improved system can better control latency for real-time events.

Detailed Description Text (70):
If the DirectDSP software allocates two granules wherein one creates data and the other uses the data, a data dependency or synchronization issue is avoided by the system of "handles" by which pieces of software under Windows hand off from one piece to another. Transactions under Windows OS are essentially file-based where source and destination handles are passed from one process/thread to another to facilitate program execution. Analogy with dataflow architecture applies except that software granules are linked between a host and DSP, rather than using close-coupled dataflow hardware. Analogy with link-list processing applies except that handles, not pointers, link the granules.

Detailed Description Text (75):
To launch an object, the host runs the augmented Windows OS which determines relative loading of x86 and VSP MIPS at run-time. According to an allocation algorithm, the augmented Windows OS will either allocate the host software object to the host CPU or the corresponding VSP software object to the VSP. Meanwhile, data passes to and through system memory space according to the common data structure so that the processing site, as host or VSP, does not matter. This implies processor independence.

Detailed Description Text (77):
The wrapper acts as a scatter-gather bus master and I/O accelerator by itself that boosts throughput of a multitasking system (even without a DSP chip or core) by relieving the host of I/O chores and providing byte channeling of 32-bit Dword host data into byte-aligned 16-bit VSP word format without host or VSP intervention. The wrapper also has a memory buffer for modem, voice/telephony and audio data. With a

DSP, the VSP wrapper can "walk" the entire virtual memory space of the host memory system without host intervention thereby making the VSP a super bus master with virtual memory addressing capability beyond simple scatter-gather bus mastering. With a DSP, the VSP wrapper can further create ping-pong and circular buffers to advantageously unify the buffers currently used in modem, voice and audio applications by replacing modem, voice/telephony and audio add-in cards with the VSP circuitry.

Detailed Description Text (94):
When a memory object is allocated, a handle, rather than a pointer, is generated to identify and to refer to the memory object. The handle is used to retrieve the current address of the allocated memory object. For example, a source handle references a source memory buffer. Processing puts data in a destination memory buffer which is referenced by a destination handle. When a task needs to access the memory object, the handle for that memory object is preferably locked down. The action of locking down a memory handle temporarily fixes the address of the memory object and provides a pointer to its beginning. While a memory handle is locked, Windows cannot move or discard the memory object. After the object is accessed or the object is not in use, the object handle is then unlocked to facilitate Windows memory management.

Detailed Description Text (95):
USP utilizes this fundamental memory management scheme to make a VSP an extension of the host CPU and to share host system memory and resources. USP provides a method for the VSP to grab memory object handles. Since Windows provides OS services for ascertaining the physical addresses of memory objects when they are locked down, the VSP grabs these handles by Direct DSP software operations that obtain the physical addresses of these handles through Windows and pass them on to the VSP. With these physical addresses, the VSP accesses memory objects (e.g. via the PCI bus) with VSP acting as a super busmaster for scatter-gather DMA transactions within the entire host accessible virtual memory space. The host CPU/MMX has elaborate paging hardware on-chip for accessing 64 T bytes of virtual memory. VSP conveniently traverses the host virtual memory space as a super busmaster by using these handles (translated to physical addresses) provided by host and OS enhanced with DirectDSP operations.

Detailed Description Text (118):
In FIG. 6 of incorporated U.S. patent application Ser. No. 08/823,257, USP enhances the basic superscalar Pentium CPU by providing a third processing or execution pipe with out-of-order execution of DSPops (DSP macro operations comprised of DSP instructions) running on the VSP. An application program comprises processes (tasks) and/or threads with a series of Memory and/or I/O transactions. If the memory handles were pointers, this execution scheme resembles a processing link-list for the granules of each application. With each granule executing on a combination of the U, V pipes or the DSP pipe, the VSP constitutes a superscalar extension of the CPU/MMX with DSPops scheduled and dispatched to it via DirectDSP. The VSP can be programmed as a Scalar (SISD), Vector (SIMD), or VLIW macrostore for DSPops.

Detailed Description Text (121):
Some methods herein utilize file-based transactions under Windows OS where source and destination handles are passed from one process/thread to another to facilitate task execution. Handles resemble pointers, but they are distinct in this technology. In FIG. 7 of incorporated U.S. patent application Ser. No. 08/823,257, CPU/MMX works on source data in source memory space by obtaining a source handle. The results are then passed to destination memory space via a handle for further processing by the VSP which grabs a destination handle via DirectDSP. The VSP processing results in destination space are forwarded with a handle to the next processing stage, perhaps by the CPU/MMX and so on. If handles are thought of as pointers (once the memory objects are locked down), some embodiments create a link-list of transactions and a task is broken up into a series of system memory transactions and/or I/O transactions performed with CPU/MMX or VSP MIPs where the CPU/MMX and VSP are essentially coupled together via shared host system memory.

Detailed Description Text (123):
USP implements a software caching scheme to insert the VSP memory spaces into the host virtual memory space thereby utilizing the host's caching mechanism as well as

its own for memory accesses. The program code and data for the VSP are continually cached into the DSP core or chip from the VSP wrapper program and data space in host (system) virtual memory for execution as shown in the VSP software caching model, FIG. 9 of incorporated U.S. patent application Ser. No. 08/823,257. Since the data processed by the VSP are real-time digital signals or non-cacheable data, a software (paging) caching scheme rather than a traditional Pentium CPU caching scheme is used for the VSP. A traditional L1, L2 type of write-back or write-through cache might have the undesirable effect of cache thrashing when used with non-cacheable data. The VSP software or paging cache acts as macrostore for DSPops executed in parallel with Host CPU/MMX instructions.

Detailed Description Text (129):
Some process embodiments advantageously redirect data to where it needs to be processed, thereby redistributing system MIPs and bandwidth utilized for compression and decompression tasks. For example, DirectDSP granule allocation dispatches compressed MPEG video/audio or AC3 audio to the VSP for processing where compressed audio transfers across the system bus instead of host-decompressed video/audio. In addition, both bus bandwidth and memory utilization are less burdened if the video/audio output is further sent to codec coupled to the VSP back-end. If the host CPU were to decompress MPEG or AC3 audio, it would have to send decompressed audio output across the system bus to the codec, thereby causing more bus bandwidth utilization. Also, because of program and data alignment issues of the host CPU/MMX architecture, more memory bandwidth/utilization is required. By contrast, the VSP decompression utilizes very compact DSP program code and efficiently handles non-cacheable audio/video data. Not only does hot processing use up more data and code memory bandwidth, but also multimedia non-cacheable data will also thrash the host L1 and L2 caches, with excessive uncontrollable latency detrimental to real-time signal processing.

Detailed Description Text (130):
In FIG. 12 of the incorporated U.S. patent application Ser. No. 08/823,257, with host CPU only, MPEG tasks are sequentially executed and the CPU only devotes a portion of the real-time to each task. Therefore, the time slots outside of each task are devoted to other tasks and can be considered as "dead time" as far as the current task is concerned.

Detailed Description Text (134):
Super busmaster with scatter-gather DMA to access all (e.g., 64 Tbytes) of virtual memory space in host memory. This entails "walking" individually scattered 4K pages under Windows 9x. I/O re-direction of data for bus-independent output or re-targeting of data to different output devices.

Detailed Description Text (137):
Real-time multimedia interrupts are effectively virtualized and handled by the VSP instead of the host CPU/MMX to avoid host context switching overhead for external interrupts under Windows. Another implementation slows down an external high-frequency interrupt by splitting interrupt processing into two stages wherein the high-frequency stage is handled by the VSP with a guaranteed response time and the processed interrupt is passed on to the host if necessary as a low-frequency interrupt from the VSP. The host CPU/MMX then processes the low-freqency interrupt with a short interrupt service routine (ISR) which schedules a deferred procedural call (DPC) to finish off the processing for the external event. DPC does not interfere with the processing of other Windows threads, since the ISR is extremely short (i.e. small fixed overhead). Advantageously, other events, threads or processes are minimally locked out, thereby streamlining operations in a multi-tasking multi-threaded system and/or multiprocessor system.

Detailed Description Text (138):
Deterministic response time for real-time applications is afforded when the VSP is used to guarantee processing time to the external events/interrupts and control latency due to its processing for the most critical (high-frequency portion) part of the real-time event processing. The VSP operations blend into the Windows OS operations for optimum execution. In real time systems, latency refers to the total time that it takes the host CPU to acknowledge and handle an interrupt. Consider a time interval occupied by high-frequency VSP interrupt handling followed by

low-frequency host ISR and then non-time-critical Windows thread execution with a DPC. That time interval encompasses all operations that handle an external real-time multimedia interrupt, and can be substantially determined and controlled according to the processes of operation and architectural embodiments disclosed herein.

Detailed Description Text (139):
In general, a multi-tasking, multi-threaded OS schedules tasks more efficiently if they appear to the OS as asynchronous I/O tasks which require minimal host intervention and less "thrashing" of the host cache(s). The DirectDSP, HAL, DSP kernel and VSP arrange multimedia tasks into this form. In this way, the system is more balanced and its throughput accelerates. Asynchronous I/O is a very powerful mechanism for real-time applications where each task can queue I/O loads (tasks) and continue processing without having to either wait or respond immediately to some end-of-I/O event. Apart from minimal host intervention and less cache "thrashing", this pays enormous dividends on multi-processor systems and reduces I/O overhead on single processor systems.

Detailed Description Text (140):
The VSP acting as a super busmaster becomes an asynchronous I/O controller which not only comprehends, spans and traverses the entire host virtual memory space but also provides processing MIPs with each transfer. The VSP acts as a powerful I/O "traffic cop" that streamlines host operations and increases system throughput.

Detailed Description Text (142):
Advantageously, USP does not need an OS of its own. See FIG. 8 of incorporated U.S. patent application Ser. No. 08/823,257. Instead, USP uses Windows OS as its own OS via DirectDSP and the real-time VSP Kernel software (USP resource management is built into DirectDSP and the VSP kernel). This software architecture is both complementary and non-competing with the Windows OS. In the preemptive, multi-threaded, multi-tasking Windows OS, processes and threads are normally running at S/W IRQLs with lower priorities than the H/W IRQLs. Although threads can be raised to real-time high priority via software, they are still at or below IRQ2 (dispatch). In FIGS. 29, 30, 31 and 32, by tying a process or thread to a H/W event/interrupt (IRQ12-IRQ27), USP raises the process or thread priority to above other software (host-based) processes or threads.

Detailed Description Text (143):
Short interrupt service routines (ISRs) are used along with deferred procedural calls (DPCs) as well as I/O request packets (IRPs) to improve system latency and turnaround time. DirectDSP WDM (or DirectDSP HAL) operates at ring 0 to reduce ring transitions to ring 3 for resources. This provides software latency control for real-time applications.

Detailed Description Text (144):
Not only do VSPs efficiently handle real-time events and multimedia, they further enhance the Windows OS by virtualizing real-time Interrupts and DMAs. A VSP can even act as an MMX emulator/accelerator or a WDM accelerator accelerating the Windows OS.

Detailed Description Text (149):
In-place processing of real-time stream I/O data for input to host memory

Detailed Description Text (208):
FIG. 125 illustrates DirectDSP as an extension of Windows DirectX where a DirectDSP HAL replaces the DirectX HAL and supports both sets of APIs. Whether an application calls DirectX or DirectDSP, the DirectDSP HAL supports them both by utilizing the COM-based interface of DirectX, DirectDSP and DirectDSP HAL. DirectDSP HAL is suitably replaced by a DirectDSP WDM to support the new Windows Driver Model (WDM), and the VSP thereby becomes a WDM accelerator. VSP accelerates ActiveX which uses either DirectX or WDM. Indirectly, USP accelerates Windows OS and enhances the ultimate system throughput when processing real-time newmedia applications.

Detailed Description Text (229):
FIG. 33 depicts the operation of the real-time DSP Kernel software in conjunction with the hardware. The diagram illustrates multiple audio out tasks, an audio in

task, and a generic DirectDSP task running concurrently. The following items are represented in the diagram:

Detailed Description Text (243):
Standard PCI I/O examples are PCI Read, PCI Write, Program Read and so forth. Message queue I/O involves processing of the DSP and host message queues. Stream I/O involves the processing of streams as described later herein. In stream processing, the DSP performs scatter-gather DMA operations. The PCI bus master ISR when processing a PCI request calls the required function via a pointer to a function in a PCI request packet. A task specifies the function to be called by inserting the desired function address into this function pointer. The method and architecture allow application specific I/O functions other than those supplied by the permit DSP kernel to be performed by the PCI bus master ISR as long as the functions conform to PCI bus master ISR coding requirements. In this way, an unlimited set of PCI I/O processing types are advantageously provided.

Detailed Description Text (252):
For example, wave synthesis might require a 64 sample frame while AC-3 would require a 256 sample frame. Audio out ping and pong buffers correspond in size to each application frame size. For latency reasons, the SC Xmt buffer size might be only 16 samples. If this is the case, the audio out mixer executes four times before the wave synthesis task is notified of an empty audio out buffer. Because the size of the audio out buffers can be set by the task and is therefore variable, the audio out mixer keeps track of a pointer to valid data and, when the pointer is at the end of a buffer, switches to the next valid buffer, either the ping or pong. Once emptied, the audio out mixer sets a semaphore to notify the appropriate task that the buffer has been emptied. The task then executes to process another frame of data and fill the audio out ping or pong buffer.

Detailed Description Text (253):
In FIG. 42, a link-list of valid Audio Out buffers is kept for the audio out mixer to process. This list consists of pointers to each audio out buffer structure. The mixer processes this list to mix each audio out buffer into the SC Xmt buffer. If the mixer gets to the end of, say, a ping buffer, and the pong buffer is not valid, then the mixer removes the audio out buffer from the list to be processed. When the task finally gets around to updating the audio out buffer, it again places the audio out buffer into the audio out buffer list processed by the mixer. An audio out buffer will cease to be mixed into the SC Xmt buffer when it is removed from the audio out buffer list. This cuts down on the amount of overhead required in the audio out mixer.

Detailed Description Text (270):
ppv: Pointer points to a buffer where the interface is expected to be returned

Detailed Description Text (275):
pWave: Pointer points to the object structure

Detailed Description Text (280):
ppv: Pointer points to a buffer where the interface is expected to be returned

Detailed Description Text (284):
pIDDspWave: Pointer points to the object structure

Detailed Description Text (288):
pIDDspWave: Pointer points to the object structure

Detailed Description Text (292):
pIDDspWave: Pointer points to the object structure

Detailed Description Text (294):
lpParam: Pointer points to the parameter list

Detailed Description Text (298):
pIDDspWave: Pointer points to the object structure

Detailed Description Text (299):
lpdwVolume: Pointer points to the volume value to be set

Detailed Description Text (303):
pIDDspWave: Pointer points to the object structure

Detailed Description Text (305):
lpdwposition: Pointer points to the buffer where position will be returned.

Detailed Description Text (309):
pIDDspWave: Pointer points to the object structure

Detailed Description Text (314):
pIDDspWave: Pointer points to the object structure

Detailed Description Text (319):
pIDDspWave: Pointer points to the object structure

Detailed Description Text (321):
IpBuffer: Pointer points to the data buffer

Detailed Description Text (325):
pIDDspWave: Pointer points to the object structure

Detailed Description Text (327):
lpBuffer: Pointer points to the data buffer

Detailed Description Text (331):
pIDDspWave: Pointer points to the object structure

Detailed Description Text (339):
ppv: Pointer points to a buffer where the interface is expected to be returned

Detailed Description Text (344):
ppv: Pointer points to a buffer where the interface is expected to be returned

Detailed Description Text (348):
pIDDspMAudio: Pointer points to the object structure

Detailed Description Text (352):
pIDDspMAudio: Pointer points to the object structure

Detailed Description Text (356):
pIDDspMAudio: Pointer points to the object structure

Detailed Description Text (358):
pBuffer: Pointer points to the memory block to be locked

Detailed Description Text (364):
pIDDspMAudio: Pointer points to the object structure

Detailed Description Text (366):
pBuffer: Pointer points to the memory block to be unlocked(must use the pointer returned from LockMemory)

Detailed Description Text (370):
pIDDSpMAudio: Pointer points to the object structure

Detailed Description Text (373):
pParam: Pointer points to the parameter list

Detailed Description Text (377):
pIDDspMAudio: Pointer points to the object structure

Detailed Description Text (379):

pBuffer: Pointer points to the buffer

Detailed Description Text (384):
pIDDspMAudio: Pointer points to the object structure

Detailed Description Text (386):
pSrcBuffer: Pointer points to the source buffer

Detailed Description Text (388):
pDstBuffer: Pointer points to the destination buffer

Detailed Description Text (393):
pIDDspMAudio: Pointer points to the object structure

Detailed Description Text (394):
lpdwPosition: Pointer points to the buffer where stopped position is to be returned

Detailed Description Text (398):
pIDDspMAudio: Pointer points to the object structure

Detailed Description Text (405):
ppv: Pointer points to a buffer where the interface is expected to be returned

Detailed Description Text (410):
ppv: Pointer points to a buffer where the interface is expected to be returned

Detailed Description Text (414):
pIDDspAcm: Pointer points to the object structure

Detailed Description Text (418):
pIDDspAcm: Pointer points to the object structure

Detailed Description Text (422):
pIDDspAcm: Pointer points to the object structure

Detailed Description Text (424):
pBuffer: Pointer points to the memory block to be locked

Detailed Description Text (430):
pIDDspAcm: Pointer points to the object structure

Detailed Description Text (432):
pBuffer: Pointer points to the memory block to be unlocked(must use the pointer returned from LockMemory)

Detailed Description Text (436):
pIDDspMAudio: Pointer points to the object structure

Detailed Description Text (439):
pParam: Pointer points to the parameter list

Detailed Description Text (443):
pIDDspAcm: Pointer points to the object structure

Detailed Description Text (445):
pSrcBuffer: Pointer points to the source buffer

Detailed Description Text (447):
pDstBuffer: Pointer points to the destination buffer

Detailed Description Text (452):
pIDDspAcm: Pointer points to the object structure

Detailed Description Text (484):
Assuming the StartIo routine finds that the transfer can be done by a single DMA

operation, the StartIo routine calls IoAllocateAdapterChannel with the entry point of the driver's AdapterControl routine and the IRP. When the system DMA controller is available, an IRP next calls the AdapterControl routine DDAdapterControl to set up the transfer operation. The AdapterControl routine calls IoMapTransfer with a pointer to the buffer, described in the MDL at Irp.fwdarw.MdlAddress, to set up the system DMA controller. Then, the driver programs its device for the DMA operation and returns. When the device interrupts to indicate its transfer operation is complete, the driver's ISR DDInterruptService stops the device from generating interrupts and calls IoRequestDpc which executes another IRP to queue the driver's DpcForIsr routine DDDpcForisr to complete as much of the transfer operation as possible at a lower hardware priority (IRQL).

Detailed Description Text (508):
The PCI bus master block offers single cycle and burst transfers via memory space, as well as I/O space; the transfers include all types of PCI transfers. With the byte channeling hardware of FIGS. 53 and 54, the PCI bus master transfers data (1 byte up to the size of ASIC RAM) from any byte addressable host memory location to any byte memory location in the ASIC RAM, starting and stopping data on any byte boundary.

Detailed Description Text (526):
The C5x boatloads from data loaded into the DPRAM. When the VxD initializes the system, data is loaded into the DPRAM by the PCI host via slave data transfers. Then the C5x is taken out of reset. The C5x then reads global data location FFFFh. This address FFFFh maps to DPRAM. The data at this location tells the C5x what type of bootload to perform and an address space from which to start loading. The bootload program then loads the C5x initialization code from the DPRAM and starts executing the downloaded code. The DPRAM address pointers are initialized to facilitate the bootload sequence.

Detailed Description Text (532):
The registers available to the host are accessible via either I/O or memory space.

Detailed Description Text (544):
TI C5x and C54x DSP's, have a unified memory architecture (64K on-board SRAM accessible via either program space or data space). In FIG. 51A, ASIC 1720 drives control signals such as chip enable for the SPAMs 3330.21 and 3330.22. Advantageously, with the ASIC driving the SRAM control signals, the DPRAM can exist in the same memory space as the SRAM. The ASIC simply need not turn on the SRAM signals if the address being accessed is currently pointed to ASIC DPPAM. FIG. 51B shows a SRAM read timing example.

Detailed Description Text (554):
Base address 0 requests 32 I/O addresses for access to the registers found in a "PCI Host Accessible Registers" area of FIG. 54I. Base address 1 mirrors these I/O addresses in memory space. The memory space is non-prefetchable in this embodiment.

Detailed Description Text (556):
PCI registers include a status and command register 0x08 dword tabulated here. Two upper status bytes therein are only reset (not set) by host. 2 lower command bytes include I/O space, memory space and bus master bits set by BIOS software.

Detailed Description Text (557):
A PCI miscellaneous control register 0x40 is tabulated next. The software reset bit 31 in miscellaneous control register 0x40 resets the entire ASIC with the exception of the base address registers and the EEPROM values (Subsystem ID, Subsystem Vendor ID, Max Latency, and Min Grant) in the configuration space. PCI bus mastering is turned off, but the EEPROM is not reread. The software reset bit is static and does not reset itself to 0 if it is set to 1. The PCI host sets the bit to 1 to reset the ASIC, then resets the bit to 0 to take it out of reset.

Detailed Description Text (558):
FIG. 54I depicts the PCI host accessible registers, which are accessible both in IO space and memory space. An asterisk (*)indicates a register which has write capability shared and controlled by PCI blocked and DSP.

Detailed Description Text (563):
A 32 bit host memory space pointer 0x08 is initialized by the host and points to a location in system memory that the DSP can access. In this embodiment, the memory space allocated by the host is not less than 4K bytes, so this register implements the bits (31:12).

Detailed Description Text (580):
Buffer size and address registers (0x20-0x30). See definitions of these under ASIC DPRAM partitions (0x60-0x69) description later hereinbelow. The address and buffer size in these registers is in terms of host byte addresses for the ASIC DPRAM and the number of bytes in each buffer. The values start out as C5x 16 bit words, but are converted in hardware to byte addresses and byte buffer sizes. The ping and pong buffers are set up as contiguous memory spaces purposely. Since the ping is always assumed to be the first buffer used, a counter can be set up with the ping buffer address. Once the ping buffer has been read or written, the address counter can continue to count. This new count value will roll right into the pong buffer space without having to reload the counter. The address counter is suitably the maximum size to point to any location in memory. The buffer address is held in bits 10:0 and the buffer size in bits 9:0 of their respective registers.

Detailed Description Text (626):
In FIG. 54, the I/F to the DPRAM includes a counter for each "byte column" in the memory. These counters act as the pointers into memory. By incrementing these counters when the memory is enabled, only the desired bytes are put into the DPRAM, and they are correctly positioned by the shifter.

Detailed Description Text (730):
The bootload process for the C54x resembles the process for the C5x with one exception. The C54x makes its initial read at address FFFFh from IO space (since the C54x does not have global memory). To accommodate both processors, the data stored in the DPRAM at the highest address is routed through the IO register output muxes, because the IO strobe signal controls the final mux stage of the data to the DSP. FIG. 68A illustrates this process.

Detailed Description Text (756):
DPRAM DS/PS address pointer register 0x5E allows the C5x to program which section of memory corresponds to the ASIC DPRAM. These bits are be compared to the upper address bits of the C5x address for reads and writes to the DPRAM.

Detailed Description Text (760):
The buffer addresses and locations show an example in which the buffer size for the voice codec buffers is 128 words each. VCX=voice codec transmit. VCR=voice codec receive. The stereo codec has 256 words, for example, (100h) for each ping or pong buffer. The DMA buffer shares memory space with an optional MIDI buffer starting at address 0x100. The DMA buffer size is 128 words for each ping and pong buffer, while the MIDI buffer size is 16 words per buffer. Without DSP intervention, the DMA will not be used, so the DMA buffer and the MIDI buffer do not conflict. The host can perform the voice codec or stereo codec functions without booting the DSP because these default buffer sizes are provided.

Detailed Description Text (931):
Further processes of system 200 include H.245 call control process 235 coupled to telephone book pages services such as user location 240 and directory services 245. ActiveX Movie process 250 is supported by H.263 video (POTS videoconferencing) and G.723 Audio. A real-time protocol RTP process 260 couples to platform 205.

Detailed Description Text (937):
Further in FIG. 3, a VSP-improved integrated circuit 345 of FIG. 17 incorporating the DSP 1730 and wrapper 1720 circuitry of FIG. 50 runs VSP kernel software according to USP shared memory model. Chip 345 also includes PCI master/slave coupling to PCI bus 330. Chip 345 has its memory size and pinout tailored for 3D geometry slope/setup, and MPEG compression/decompression algorithms. Advantageously, CPU 315 is relieved of burden of much of these calculations, and freed from much time-consuming MMX context switching latency.

Detailed Description Text (942):
Chip 510 has its memory size and pinout tailored for 3D graphics and geometry slope/setup, MPEG compression/ decompression algorithms, and/or 3D audio. Advantageously, CPU 315 is relieved of burden of much of these calculations, and freed from much time-consuming MMX context switching latency. Another embodiment of chip 510 integrates blocks 520 and 525 together with advantageously low real estate and reduced pinout, and PCI/PCI block 530 is a separate chip.

Detailed Description Text (946):
A wrapper/DSP chip 720 for 3D graphics has its wrapper adapted for AGP instead of PCI, and chip 720 is coupled to AGP interface 118 of chip 108. A real-time RT link couples 3D chip 720 to a LAN/IEEE 1394 wrapper/DSP chip 725. A IEEE 1394 link couples chip 725 to a wrapper/DSP 730 which has its wrapper adapted for serial bus interface instead of PCI. By contrast chip 725 has a PCI master/slave interface built into it and coupled to PCI bus 330, as indicated a line from chip 725 to PCI bus 330. (The reader should thus note that these diagrams use a compressed style of representation to show the types of interfaces used. Further in the compressed style, each legend suggests pinout embodiment type as well as software appropriate to the legend.)

Detailed Description Text (950):
North of PCI bus 330, a wrapper/DSP chip 820 has a main memory 325 coupled to a single chip 810 which has pinout for memory bus to DRAM, for PCI bus 330, for a real-time RT port, and for TV tuner and RAM and DAC integrated onto the same single chip. By using 0.25 micron-or-less CMOS and mixed signal technology, this chip provides attractively compact package. Another embodiment partitions off the TV-tuner/RAMDAC block 890 from chip 810.

Detailed Description Text (951):
In chip 810, block 850 is a Host CPU at P5, P6, P7, P68, P8 or other level of functionality, enhanced as the skilled worker may desire with multimedia extensions in instruction set. An embedded L2 cache 855 is coupled with Host 850. Dotted lines indicate that this cache 855 is omitted in another embodiment. Block 850 is coupled to a north bridge block 860 having PCI bridge/MCU, embedded L2/L3 DRAM cache, AGP interface 862 and 3D geometry/MMX accelerator wrapper DSP 864. A block 870 on the same chip 810 has wrapper/DSP for 3D graphics and audio, an AGP interface coupled to AGP 862, and a real-time RT interface. Note that the two interfaces are on the same single chip, and another embodiment may vary the interfaces or eliminate them. The block 870 wrapper/DSP is coupled to an embedded frame buffer 880 which in turn couples to TV tuner/RAMDAC 890.

Detailed Description Text (952):
An additional wrapper/DSP 820 has at least three interfaces: 1) RT interface coupled to block 870 of chip 810, 2) PCI master/slave interface coupled to PCI bus 330 and 3) cardbus interface to cardbus slots. The wrapper/DSP virtualizes super-real-time hub functions and cardbus data processing such as audio and modem processing advantageously architecturally near the cardbus slots even when chip 840 is also present.

Detailed Description Text (962):
In FIG. 18, interrelated improved processes called DirectDSP, DirectDSP HEL (host emulation), DirectDSP HAL (hardware abstraction layer), and VSP Kernel (DSP Real-Time Kernel)

Detailed Description Text (973):
In step 2424, the host runs at least some of the operating system code including an allocation algorithm as described in connection with FIGS. 24A and 24B. A step 2428 determines whether a DSP object code granule is allocated to host by allocation in step 2424, and if no, operations proceed to execute the DSP object granule on DSP in a step 2436. If yes in step 2428, and a step 2432, allocation to host in step 2424 causes the host object code granule for the same function to instead be executed on the host in a step 2438. Operations loop back to step 2424 to continually allocate functions or pieces in application software to the host or DSP to dynamically balance the system 100. In a step 2442, execution of a granule either on host or DSP

sends results to a common data structure or shared memory space in the main DRAM memory 110 (see FIG. 94 and FIGS. 57A-F).

Detailed Description Text (981):
3A) total bandwidth of (each) bus such as CPU bus, mezzanine bus (e.g., PCI), I/O bus (ISA, USB, AGP, IEEE 1394, Zoom Video, real-time RT bus, etc.)

Detailed Description Text (984):
4A) total main memory space and each outlying memory space available

Detailed Description Text (985):
4B) total usage of main memory space and each outlying memory

Detailed Description Text (986):
4C) available main memory space and available space in each outlying memory. 4C=4A minus 4B.

Detailed Description Text (998):
If the application has been launched on the same system before, the foregoing operations are already completed, and operations instead commence at a BEGIN 2480 and proceed to a step 2484. Step 2484 obtains and calculates the real-time loading of the DSP, Host CPU, memory, and I/O, such as described in connection with Table 0 above. A succeeding step 2488 executes allocation logic, as discussed earlier hereinabove in connection with this FIG. 24B.

Detailed Description Text (1032):
Turning to the subject of handles, a function call in Direct DSP translates the handle logical address to physical address when pages are locked down. The 32-bit physical address is sent to VSP. If VSP executes granule in DSP code then it uses the physical address. On the other hand, if host emulation is used, then host code recognizes the logical address. The driver code is transparent to APP.exe. VSP 1720, 1730 advantageously walks the entire virtual memory space of the host (e.g. 64 terabytes) with no virtual paging logic implemented on VSP at all.

Detailed Description Text (1034):
In FIG. 28, memory spaces representing a shared memory model utilized in embodiments of processes, devices and systems illustrate the source handle, destination handle and VSP handle arrangement.

Detailed Description Text (1036):
In FIG. 29, interrupt levels are utilized in connection with hardware interrupts and deferred procedure calls (DPCs) in process, device and system embodiments as shown. Notice use of both IRQs and DPCs. Priority raising occurs at arrow 2910 and 2920. Priority falls as indicated by arrows 2930, 2940 and 2950. This advantageous behavior used by the DirectDSP HAL and VSP Kernel is also depicted in FIGS. 30 and 32.

Detailed Description Text (1037):
In FIG. 30, interrupt levels over time utilized in connection with hardware interrupts and deferred procedure calls (DPCs) in process, device and system embodiments

Detailed Description Text (1038):
In FIG. 31, interrupt levels in real-time classes 3110 and dynamic classes 3120, 3130 and 3140 are shown.

Detailed Description Text (1047):
In FIG. 39, host memory program and data spaces (at top) and DSP memory space (at bottom) represent an example of handles and data structures in the shared memory model of FIG. 36 utilized in FIG. 33. DWDSPOBJADDR is a pointer derived from a handle when the memory space is locked down. In Windows a handle is an identifier for a 32-bit address. When the memory is locked down, the full address becomes known at that time and becomes a pointer.

Detailed Description Text (1056):

In FIG. 48, a system 4800 has a VSP-based combined audio controller 4830, 4840, 4850 and modem 4860. AC link and control block 4850 links to AC97 audio/modem codec 4870. Sample rate converter and mixer block 4820 is also included. VSP advantageously virtualizes these functions.

Detailed Description Text (1062):
Hitherto, modem, voice, stereo audio, and other interfaces have often been implemented on respective add-in cards with respective software drivers and respective slave bus interfaces. Such system architecture has burdened OEM and business and consuming public with space requirements and financial expense. CPU MIPS are expended on the numerous slave transactions as well. Advantageously, FIG. 50 embodiment shows how a single bus master serves all these application hardwares, and relieves the Host of the extra burden of communicating to slave circuits, reducing Host I/O MIPS significantly. Without the DSP 1730 connected, wrapper ASIC 1720 provides basic scatter-gather bus master capability for traversal of some memory spaces. With the DSP 1730 in place, wrapper ASIC 1720 and DSP 1730 together provide super-bus-mastering to access the entire memory space in the system, and in Host terms the entire virtual memory space accessible by Host.

Detailed Description Text (1091):
DSP-accessible ASIC control registers 5020 provide respective DMA address and DPRAM address lines to a mux 5355 in a PCI interface control block 5340. This mux selects the appropriate address information in response to a control line for type of transfer coupled and then supplies that information to the rest of PCI interface control block 5340 for memory accesses, and to the byte channeling block 5310 for address translation and appropriate byte accessing of DPRAM 3330.1.

Detailed Description Text (1092):
PCI block 5010 couples data from PCI bus 330 via a flip-flop in block 5010 to data_in lines to byte channeling unit 5310. PCI block 5010 couples byte enables PCI bus 330 via lines C_BE to byte channeling unit 5310. Slave addresses, data-in, and byte enables are coupled from PCI block 5010 to PCI slave registers in block 5018. Slave addresses are also coupled to PCI interface control block 5340.

Detailed Description Text (1103):
In FIG. 54I, PCI host accessible registers start at Base Address BA0 in PCI I/O space and replicated start at Base Address BA1 in PCI memory space of FIG. 128, (BA0, BA1 defined in PCI Configuration Register 0x10, 0x14). FIG. 54I further indicates address offset decodes and read or read/write circuits associated with those PCI host accessible registers in the wrapper ASIC of VSP.

Detailed Description Text (1117):
In FIG. 57F, host main DRAM memory has a DSP message queue and a host message queue with host manipulated head and tail pointers on the left side, and DSP manipulated head and tail pointers on the right side.

Detailed Description Text (1128):
In FIG. 67, wrapper ASIC DPRAM memory space for DSP bootload purposes, the memory space pointed to by an SRC address of FIG. 70

Detailed Description Text (1149):
In FIG. 84, VSP-graphics/video chip 8440 and VSP-comm-audio-MPEG-RThub-cardbus chip 8410 are respectively coupled by real-time private bus RT 8430. Chip 8440 is supported by frame buffer 8450 coupled thereto. Chip 8410 bidirectionally communicates IEEE 1394 serial data via an IEEE 1394 PHY (physical layer) chip 8420 for video capture, for example.

Detailed Description Text (1150):
In FIG. 85, chip 8510 integrates on a single chip a wrapper/DSP that virtualizes 3D audio, geometry, RT hub and cardbus controller functions. Chip 8510 is coupled to a device bay for insertion and extraction of external peripherals, and/or to cardbus slots Slot1 and Slot2. A real-time RT video link 8430 couples to a graphics/video chip such as 1120 of FIG. 11. An IEEE 1394 serial bus PHY chip 8420 mediates video capture and couples to chip 8510.

Detailed Description Text (1161):
In FIG. 95 is a process diagram or method-of-operation diagram emphasizing DSP task object structure in the shared memory model of FIG. 94 in system embodiments. A process of operating a computer system having a storage holding an operating system and an application program, a first processor having an instruction set, and a second processor having a different instruction set, has these steps among others: 1) running the first processor to determine whether a part of the application shall be run on the first processor or the second processor and then establishing a second processor object if that part shall be run on the second processor, and otherwise not establishing the second processor object. If run on second processor, a next step 2) runs at least some of the operating system on the first processor so that the first processor sets up for at least part of the application program at run time at least one second processor object. The second processor object is suitably established by using the data structures and VSPOBJECT listing earlier hereinabove to define and then lock down areas in the shared memory space. A step 3) concurrently runs the second processor to access the second processor object and thereby determine operations for the second processor to access second processor instructions for the part of the application program and data to be processed according to the second processor instructions. Then a step 4) runs the second processor to process the data according to said second processor instructions. The second processor object includes information defining a task type, a pointer to the data to be processed, a pointer to scratch space, a buffer in/out type information, buffer dimensions information, and page list information, among other advantageous selections. A DSP second processor and an OS with x86 first processor code for IBM-compatible computer are used in some embodiments.

Detailed Description Text (1190):
In FIG. 121 an application opens and calls a function from the DirectDSP API set of functions 12024. Proceeding from DirectDSP HAL entry point 12090, a decision step 12110 determines whether the function is desired to run on host or DSP, such as indicated by the default entry host/DSP default allocation for the function in FIG. 24A. Predetermined preferences are thus stored in a lookup table, or alternatively program code provides the preference information. If default entry of DSP in step 12110, then operations proceed to a decision step 12115 to determine whether DSP MIPS are available for the whole task. A real-time MIPS usage counter or register indicates the available VSP MIPS. If yes in step 12115, then a step 12120 loads (in case of DLL) and executes the VSP object code granules for the task on the VSP. If no in step 12115, then a decision step 12125 determines whether granules of the task are required and MIPS available. If yes in step 12125, then a decision step 12130 determines whether partial execution is supported on host. If yes in step 12130, then step 12135 executes partial task emulation and sets a grant flag for a step 12170 described later hereinbelow. If "Host" in step 12110, or no in either step 12125 or in step 12130, then operations proceed to a decision step 12140 to determine whether DirectX host emulation is available. If no in step 12140 then a decision step 12150 determines whether DirectDSP host emulation is available. If yes in step 12140 or no in step 12150, then an exit step 12145 has the host emulate the function by DirectX emulation, and the DSP does not execute the function. If yes in step 12150, then a decision step 12160 determines whether whole task emulation is required, and if no in step 12160, then a decision step 12170 determines whether partial task is supported on DSP. If either yes in step 12160 or no in step 12170, then a step 12175 executes DirectDSP host emulation to run or emulate the whole task on the host. If yes in step 12170, then step 12135 executes partial task emulation on the host and the rest of the task on the VSP.

Detailed Description Paragraph Table (7):
Source Filter CLASS CsourceFilter: public CSource //base class , public CPersistStream , public ISpecifyPropertyPages { public: static CUnknown *CreateInstance(LPUNKNOWN lpUnk, HRESULT *phr); .about.CSourceFilter ( ); DECLARE_IUNKNOWN; STDMETHODIMP NonDelegatingQueryInterface(REFIID riid, void **ppv); // --- ISpecifyPropertyPages --- // return our property pages STDMETHODIMP GetPages(CAUUID * pPages); // setup helper LPAMOVIESETUP_FILTER GetSetupData ( ); // --- IPersistStream Interface STDMETHODIMP GetClassID(CLSID *pClsid); CAudioObject *m_MidiSeq; // the class reading audio data HWND m_hWndPropertyPage; // hWnd of the PropertyPage private: // it is only allowed to to create these objects with CreateInstance CSourceFilter(LPUNKNOWN lpunk, HRESULT *phr); // When the format

```
changes, reconnect . . . void CMIDISourceFilter::ReconnectWithNewFormat (void); };
Source Output Pin CLASS CAudioStream: public CsourceStream //base class { public:
CAudioStream(HRESULT *phr, CMIDISourceFilter *pParent, LPCWSTR pPinName);
.about.CAudioStream( ); BOOL ReadyToStop(void) {return FALSE;} // stuff an audio
buffer with the current format HRESULT FillBuffer(IMediaSample *pms); // ask for
buffers of the size appropriate to the agreed media type. HRESULT
DecideBufferSize(IMemAllocator *pIMemAlloc, ALLOCATOR_PROPERTIES *pProperties); //
verify we can handle this format HRESULT CheckMediaType(const CMediaType
*pMediaType); // set the agreed media type HRESULT GetMediaType (CMediaType *pmt);
// resets the stream time to zero. HRESULT OnThreadCreate(void); HRESULT
OnThreadDestroy(void); HRESULT OnThreadStartPlay(void); HRESULT Active (void);
HRESULT Inactive (void); private: // Access to this state information should be
serialized with the // filters critical section (m_pFilter-->pStateLock( )) CCritSec
m_cSharedState; // use this to lock access to // m_rtSampleTime and m_AudioObj //
which are shared with the worker // thread. CRefTime m_rtSampleTime; // The time to
be stamped on each // sample. CAudioOBject *m_AudioObj; // the current midi object
BOOL m_fReset; // Flag indicating the re-start }; Transform Filter CLASS
CAudioTransformFilter: public CTransformFilter, //base class public
ISpecifyPropertyPages //needed for a //property page { public: // // --- Com stuff
--- // static CUnknown *CreateInstance(LPUNKNOWN, HRESULT *); STDMETHODIMP
NonDelegatingQueryInterface(REFIID riid, void ** ppv); DECLARE_IUNKNOWN; // // --
CTransform overrides --- // HRESULT Receive (IMediaSample *pSample); HRESULT
CheckInputType(const CMediaType* mtIn); HRESULT CheckTransform(const CMediaType*
mtIn, const CMediaType* mtOut); HRESULT DecideBufferSize(IMemAllocator * pAllocator,
ALLOCATOR_PROPERTIES * pProperties); HRESULT StartStreaming ( ); HRESULT
StopStreaming ( ); HRESULT SetMediaType(PIN_DIRECTION direction,const CMediaType
*pmt); HRESULT GetMediaType(int iPosition, CMediaType *pMediaType); HRESULT
EndOfStream(void); HRESULT EndFlush(void); CAudioTransformFilter(TCHAR *pName,
LPUNKNOWN pUnk, HRESULT *pHr); .about.CAudioTransformFilter( ); // setup
LPAMOVIESETUP_FILTER GetSetupData( ); private: // Serialize access to the output pin
long m_FrameSize; // Frame input size (bytes) long m_FrameSizeOutput; LPBYTE
m_lpStart; LPBYTE m_lpCurr; LPBYTE m_lpEnd; BOOL m_bPayloadOnly; enum
{MAX_FRAMES_PER_OUTPUT_SAMPLE = 4}; enum {AUDIO_BUFF_SIZE = (1024 * 8)}; DWORD
m_dwCtrl; AudioCtrl m_AudioControl; CAudioDecoder *m_pAudioDecoder; // class
actually does decoding CRefTime m_TimePerFrame; CRefTime m_TimeAtLastSyncPoint;
CRefTime m_TimeSinceLastSyncPoint; int m_FreqDiv; int m_PrefChan; int m_Quality; int
m_QuarterInt; int m_WordSize; BYTE m_Buffer[AUDIO_BUFF_SIZE]; void
ProcessDiscontiuity(IMediaSample *pSample); void ProcessSyncPoint(IMediaSample
*pSample, BYTE *pSrc); HRESULT DeliverSample(IMediaSample *pOutSample, CRefTime
&TimeDecoded, int iSampleSize); void ResetAudioDecoder( ); BOOL LookForSyncWord( );
int Padding( ); void GetNextPacketChunk(LPBYTE &lpPacket, long &LenLeftInBuffer,
long &LenLeftInPacket; CRefTime m_tStop; MPEG1WAVEFORMAT m_Format; public:
LPMPEG1WAVEFORMAT get_format( ) {return &m_Format;} }; // data structure holds audio
control information. struct AudioCtrl { // // Output Frame Buffer // DWORD
dwOutBuffUsed; DWORD dwOutBuffSize; DWORD dwMpegError; LPBYTE pOutBuffer; // //
Frame decoder control // DWORD dwCtrl; // // Input buffer fields // DWORD
dwNumFrames; LPBYTE pCmprRead; LPBYTE pCmprWrite; }; Transform Input Pin CLASS
CAudioTransInputPin: public CTransformInputPin //base class { public:
CAudioTransInputPin (TCHAR *pName, CAudioTransformFilter *pFilter, HRESULT *phr,
LPCWSTR pPinName); .about.CAudioTransInputPin ( ); HRESULT CheckMediaType(const
CMediaType *pMediaType); HRESULT DecideAllocator(IMemInputPin *pPin, IMemAllocator
**ppAlloc); private: CAudioTransformFilter *pTransFilter; }; Transform Output Pin
CLASS C_AXDSP_TransOutputPin: public CTransformOutputPin //base class { public:
C_AXDSP_TransOutputPin (TCHAR *pName, C_AXDSP_TransformFilter *pFilter, HRESULT
*phr, LPCWSTR pPinName, int PinNumber); .about.C_AXDSP_TransOutputPin ( );
STDMETHODIMP NonDelegatingQueryInterface(REFIID riid, void **ppv); STDMETHODIMP
EnumMediaTypes(IEnumMediaTypes **ppEMediaType); HRESULT CheckMediaType (const
CMediaType *pMediaType); HRESULT SetMediaType(const CMediaType *pMediaType); HRESULT
GetMediaType(int iPosition, CMediaType *pMediaType); HRESULT BreakConnect ( );
HRESULT CheckConnect (IPin *pPin); HRESULT CompleteConnect (IPin *pPin); HRESULT
DecideAllocator(IMemInputPin *pMemIPin, IMemAllocator **ppMA); HRESULT
DecideBufferSize(IMemAllocator *pMemAlloc, ALLOCATOR_PROPERTIES *pProperty); HRESULT
Deliver(IMediaSample *pMD); HRESULT DeliverEndOfStream( ); HRESULT
DeliverBeginFlush( ); HRESULT DeliverEndFlush( ); STDMETHODIMP Notify(IFilter
*pFilter, Quality q); private: C_AXDSP_TransformFilter *m_pTransFilter; cPosPassThru
```

*m_pPosition; //other data members --- TODO }; Audio Rendering Filter CLASS CAudioRenderer: public CBaseRenderer //base class , public ISpecifyPropertyPages { public: // // Constructor and destructor // static CUnknown *CreateInstance(LPUNKNOWN, HRESULT *); CAudioRenderer(TCHAR *pName, LPUNKNOWN pUnk, HRESULT *phr); .about.CAudioRenderer( ); // // Implement the ISpecifyPropertyPages interface // DECLARE_IUKNOWN STDMETHODIMP NonDelegatingQueryInterface (REFIID, void **); STDMETHODIMP GetPages(CAUUID *pPages); // setup helper LPAMOVIESETUP_FILTER GetSetupData( ); CBasePin *GetPin (int n); // Override these from the filter and renderer classes HRESULT Active( ); HRESULT BreakConnect( ); HRESULT CompleteConnect(IPin *pReceivePin); HRESULT SetMediaType(const CMediaType *pmt); HRESULT CheckMediaType(const CMediaType *pmtIn); HRESULT DoRenderSample(IMediaSample *pMediaSample); void PrepareRender( ); HRESULT OnStartStreaming( ); HRESULT OnStopStreaming( ); HRESULT OpenAudioDevice( ); HRESULT CloseAudioDevice( ); public: CAudioMemAllocator m_AudioAllocator; //Our allocator CAudioTrnasInputPin m_InputPin; //IPin based interfaces CMediaType m_mtIn; //Source connection media type CAudioCtrl m_AudioCtrl; // CAudioReferenceClock m_MidiClock; //Audio clock HWND m_hwnd; //Control window handle HANDLE m_devhandle; //Audio device handle }; // Memory allocator CLASS CAudioMemAllocator: public CBaseAllocator { CBaseFilter *m_pFilter; // Delegate reference counts to CMediaType *m_pMediaType; // Pointer to the current format LPBYTE m_pBuffer; // combined memory for all buffers protected STDMETHODIMP SetProperties (ALLOCATOR_PROPERTIES* pRequest, ALLOCATOR_PROPERTIES* pActual); // // Call ReallyFree to free memory // void Free(void);

Detailed Description Paragraph Table (8):
// called from the destructor (and from Alloc if changing size/count) to // actually free up the memory void ReallyFree(void); // overriden to allocate the memory when commit called HRESULT Alloc(void); public: CAudioMemAllocator(CBaseFilter *pFilter,TCHAR *pName,HRESULT *phr); .about.CAudioMemAllocator( ); STDMETHODIMP_(ULONG) NonDelegatingAddRef( ); STDMETHODIMP_(ULONG) NonDelegatingRelease( ); void NotifyMediaType(CMediaType *pMediaType); void CloseAudioDevice( ); }; // Our reference clock CLASS CAudioReferenceClock: public Cunknown , public IReferenceClock , public CAMSchedule , public CCritSec { public: CAudioReferenceClock(TCHAR *pName, LPUNKNOWN pUnk, HRESULT *phr, CBaseRenderer *pRenderer); .about.CAudioReferenceClock( ); STDMETHODIMP NonDelegatingQueryInterface(REFIID riid,void ** ppv); DECLARE_IUNKNOWN /* IReferenceClock methods */ // Derived classes implement GetPrivateTime( ). This GetTime // calls GetPrivateTime and then checks so that time does not go backwards. // A return code of S_FALSE implies that the internal clock has gone backwards // and GetTime time has halted until internal time has caught up. STDMETHODIMP GetTime (REFERENCE_TIME *pTime); // When this is called, it sets m_rtLastGotTime to the time it returns. /* Provide standard mechanisms for scheduling events */ /* Ask for an async notification that a time has elapsed */ STDMETHODIMP AdviseTime (REFERENCE_TIME baseTime,// base reference time REFERENCE_TIME streamTime,// stream offset time HEVENT hEvent, // advise via this event DWORD *pdwAdviseCookie// where your cookie goes }; /* Ask for an asynchronous periodic notification that a time has elapsed */ STDMETHODIMP AdvisePeriodic{ REFERENCE_TIME StartTime, // starting at this time REFERENCE_TIME PeriodTime, // time between notifications HSEMAPHORE hSemaphore, // advise via a semaphore DWORD *pdwAdviseCookie // where your cookie goes }; /* Cancel a request for notification(s) - if the notification was a one * shot timer then this function doesn't need to be called as the advise is * automatically cancelled, however it does no harm to explicitly cancel a * one-shote advise. Clients call Unadvise to clear a * Periodic advise setting. */ STDMETHODIMP Unadvise (DWORD dwAdviseCookie); /* Methods for the benefit of derived classes or outer objects */ // Overrides CAMSchedules version in order to trigger the thread if needed DWORD AddAdvisePacket( const REFERENCE_TIME & time1, const REFERENCE_TIME & time2, HANDLE h, BOOL periodic ); // GetPrivateTime( ) is the REAL clock. GetTime is just a cover for it. // Derived classes will probably override this method but not GetTime( ) // itself. // The important point about GetPrivateTime( ) is it's allowed to go backwards. // Our GetTemp( ) will keep returning the LastGotTime until GetPrivateTime( ) // catches up. virtual REFERENCE_TIME GetPrivateTime( ); /* Provide a method for correcting drift */ STDMETHODIMP SetTimeDelta ( const REFERENCE_TIME& TimeDelta ); STDMETHODIMP_(ULONG) NonDelegatingAddRef( ); STDMETHODIMP_(ULONG) NonDelegatingRelease( ); void NotifyMediaType(CMediaType *pMediaType); void SetAudioDeviceHandle(HANDLE); DWORD GetTime(DWORD dwTicks); DWORD GetTicks(DWORD msTime); protected: REFERENCE_TIME m_rtPrivateTime; // Current best

estimate of time REFERENCE_TIME m_rtLastGotTime; // Last time returned by GetTime REFERENCE_TIME m_rtNextAdvise; // Time of next advise UINT m_TimerResolution; DWORD m_PrevSamples; // Previous ticks returned from midi DWORD m_PrevSysTime; // Previous system time reference DWORD m_PrevAudioTime; // Previous audio time reference CBaseRenderer *m_AudioRenderer; // pointer to the renderer MMTIME m_mmt; DWORD m_TimeFormatFlag; DWORD m_SamplesPerSec; // Thread stuff public: void TriggerThread( ) // Wakes thread up. Need to do this if // { m_Event.Set( ); } time to next advise private: BOOL m_bAbort; // Flag used for thread shutdown CAMEvent m_Event; // Signal when its time to check advises HANDLE m_hThread // Thread handle HRESULT AdviseThread( ); // Method in which the advise thread runs static DWORD _stdcall AdviseThreadFunction(LPVOID): // Function is used to // get there }; //Audio Control Window class CAudioCtrl : public CBaseControlWindow, public CBasisAudio { protected: CBaseRenderer *m_pRenderer; // Owning sample renderer object SIZE m_Size; public: CAudioCtrl (TCHAR *pName, // Object description LPUNKNOWN pUnk, // Normal COM ownership HRESULT *phr, // OLE failure mode CCritSec *pInterfaceLock, // Main critical section CAudioRenderer *pRenderer); // Delegates locking to .about.CAudioCtrl( ); STDMETHODIMP NonDelegatingQueryInterface(REFIID riid,void **ppv); HRESULT InitWindowRegion(TCHAR *pStringName); HFONT CreateVideoFont( ); RECT GetDefaultRect( ); void GetVideoFormat(VIDEOINFO *pVideoInfo); // Pure virtual methods for the IBasicVideo interface LPTSTR GetClassWindowStyles (DWORD *pClassStyles, DWORD *pWindowStyles, DWORD *pWindowStylesEx); // // Method that gets all the window messages // LRESULT OnReceiveMessage (HWND hwnd, // Window handle UINT uMsg, // Message ID WPARAM wParam, // First parameter LPARAM lParam); // Other parameter // Implement IBasicAudio Interface STDMETHODIMP put_Volume(long lVolume); STDMETHODIMP get_Volume(long *plVolume); STDMETHODIMP put_Balance(long lBalance); STDMETHODIMP get_Balance(long *plBalance); }; Audio Rendering Input Pin CLASS CAudioRendererInputPin: public CBaseInputPin //base class { CAudioRenderer *m_pRenderer; // The renderer that owns us CCritSec *m_pInterfaceLock; // Main filter critical section public: // // Constructor // CAudioRendererInputPin{ TCHAR *pObjectName, // Object string description CAudioRenderer *pRenderer, // Used to delegate locking CCritSec *pInterfaceLock, // Main critical section HRESULT *phr, // OLE failure return code LPCWSTR pPinName); // This pins identification // // Manage our allocator // STDMETHODIMP GetAllocator(IMemAllocator **ppAllocator); STDMETHODIMP NotifyAllocator(IMemAllocator *pAllocator, BOOL bReadOnly); };

Detailed Description Paragraph Table (10):
Attribute Name Data Type Description STRUCTURE GLOBAL_DEVICE_INFO: driver global data structure shared by each device object Key ULONG Next GLOBAL_DEVICE_INFO * BusType INTERFACE_TYPE BusNumber ULONG InterruptVector ULONG InterruptRequestLevel KIRQL ShutdownRegistered BOOLEAN WaveMutex KMUTEX Dispatcher object controlling access to the device object MidiMutex KMUTEX If MIDI is defined MemType ULONG DeviceObject [ ] PDEVICE_OBJECT DeviceInUse UCHAR MidiInUse UCHAR If MIDI is defined WaveInfo WAVE_INFO HwContext DSPHWCONTEXT Hardware data Synth GLOBAL_SYNTH_INFO Synth global data MixerInfo MIXER_INFO LocalMixerData LOCAL_MIXER_DATA RegistryPathName PWSTR Registry path STRUCTURE SOUND_CONFIG_DATA: Sound card configuration data Port ULONG InterruptNumber ULONG MixerSettings [ ] MIXER_CONTROL_DATA_ITEM MixerSettingsFound BOOLEAN STRUCTURE DHBUFTABLE: VSP hardware buffer table PMdl PMDL DwBufferPhys DWORD Physical address of buffer DwBufferLinear DWORD Linear address of buffer DwBufferLength DWORD Length in bytes of buffer STRUCTURE MESSAGEQUEUE: VSP mesage queue PMQ WORD * Linear address of the pipe PMQHead WORD * Head pointer in the pipe pMQTail WORD * Tail pointer in the pipe MQSize DWORD Message queue size STRUCTURE PACKET: For packet-based DMA transfer dwPacketLength DWORD Next PACKET * Point to the next packet STRUCTURE TASKBUFFER: VSP task buffer wNumQueuedPacket WORD wBufferType WORD Source or destination buffer wBufferStarted WORD 1 if DSP started working on this buffer wNextHalf WORD dwCurPacketLength DWORD dwQuePacketLength DWORD STRUCTURE DSPHWCONTEXT: VSP hardware context DspBuffer [5] DHBUFTABLE wIOAddressCODEC WORD PortBase PUCHAR wIRQ WORD DspMQueue MESSAGEQUEUE HostMQueue MESSAGEQUEUE pObject DHOBJECT * CODECMutex KMUTEX wDspInit WORD bIntFired BYTE pDSPData WORD wDSInitialized WORD STRUCTURE DHOBJECT: VSP hardware object ReferenceCount DWORD wObjectID WORD TaskBuffer [2] TASKBUFFER piDSPContext PVOID dwiDSPContextPhysAddr DWORD bSignalFlag volatile BYTE bObjectType BYTE pMdl PMDL STRUCTURE SOUND_DMA_BUFFER: AdapterObject [2] PADAPTER OBJECT We may use 2 channels BufferSize ULONG VirtualAddress PVOID LogicalAddress PHYSICAL ADDRESS Mdl PMDL STRUCTURE SOUND_DOUBLE_BUFFER : NextHalf enum {LowerHalf = 0, UpperHalf} BufferSize ULONG BufferPosition PUCHAR StartOfData ULONG Start of valid data nBytes

ULONG Number of bytes in buffer bytesFinished ULONG Pad UCHAR Padding byte to use STRUCTURE SOUND_BUFFER_QUEUE: Control processing of device queue QueueHead LIST_ENTRY Head of the queue if Irps for writing to / reading from device. Entries are cancellable Irps. BytesProcessed ULONG Bytes put into or copied from buffers UserBufferSize ULONG UserBufferPosition ULONG UserBuffer PUCHAR Buffer corresponding to next user pIrp PIRP Pointer to the current request ProgressQueue LIST_ENTRY Wave output buffers in progress. Entries on this queue are not cancellable. STRUCTURE LOWPRIORITYMODEINFO BufferQueue SOUND_BUFFER_QUEUE SamplesPerSec ULONG BitsPerSample UCHAR Channels UCHAR WaveFormat PWAVEFORMATEX State ULONG

Detailed Description Paragraph Table (15):
Control Parameter Description Program A 32-bit address which the DSP can write Address corresponding to the DSP program space in host memory. PCI Address A second 32-bit address which the DSP can write for accesses to any PCI memory location. ASIC RAM ASIC RAM address to begin transaction. This Address value may be in either bytes or words depending on the byte/word control parameter. PCI Address Offset from the program or PCI addresses to Offset begin the PCI transaction. This value may be in either bytes or words depending on the byte/word control parameter. Xfer Count This value represents either the number of bytes or words to transfer depending on the byte/work control parameter. Start Xfer Bit allowing the DSP to begin the transfer (must automatically reset). Byte/work Bit which determines whether the Offset, Xfer Control count, and ASIC RAM address specify bytes or words. Address Determines which address pointer to use, either Selection program address or PCI address. Xfer Direction Bit which selects if a write or read is to be performed. Memory Write Selects type of PCI write to perform, either Type write or write and invalidate. Byte Pad Bit which indicates that every other byte of Enable data transferred from the PCI bus shall be padded with 0s.

Detailed Description Paragraph Table (26):
C5x I/O Space Accessible Registers Address # of Bits Register Function R/W Name 0x50 8 DSP Interrupt Mask Register R/W 0x51 8 DSP Interrupt Register R/W 0x52 12 PCI Xfr DPRAM Addr R/W dpram_addr 0x53 16 C5x Generated PCI Addr LSW R/W host_addr 0x54 16 C5x Generated PCI ADDR MSW R/W 0x55 16 PCI Macro Address Offset Register R/W addr_offset 0x56 16 PCI Macro # of Words/Bytes Register R/W num_word_byte 0x57 16 PCI Macro Control Register R/W 0x58 16 DMA SRAM Addr R/W sram_addr 0x59 16 DMA Word Count R/W word_cnt 0x5A 14 DMA Control/DMA Granularity/Delay R/W dma_gran/dma_dly 0x5B Reserved (returns "0" when read) R 0x5C 6 *Voice Codec Control Register R/W 0x5D 6 *Stereo Codec Control Register R/W 0x5E 10 DPRAM DS Pointer DPRAM PS Pointer R/W dpram_ds/dpram_ps 0x5F Reserved for (C54x) Bootland R 0x60 11 DMA Ping Address R/W dma_adr 0x61 10 DMA Buffer Size R/W dma_buf 0x62 11 Voice Codec Xmit Ping Address R/W vcx_adr 0x63 10 Voice Codec Xmit Buffer Size R/W vcx_buf 0x64 11 Voice Codec Recv Ping Address R/W vcr_adr 0x65 10 Voice Codec Recv Buffer Size R/W vcr_buf 0x66 11 Stereo Codec Xmit Ping Address R/W scx_adr 0x67 10 Stereo Codec Xmit Buffer Size R/W scx_buf 0x68 11 Stereo Codec Recv Ping Address R/W scr_adr 0x69 10 Stereo Codec Recv buffer Size R/W scr_buf 0x6A 2 PCI Slave Retry All SRAM CE register R/W 0x6B 16 PCI Diagnostic Center R 0x6C 1 PCI Int R/W 0x6D 4 C54xI/F SRAM Spd BIO Mux No-SW-WS R/W 0x6E 6 *Stereo Codec PIO Control R/W 0x6F 16 *Auto PIO Addr *Auto PIO Data R/W *Write capability shared and controlled by PCI.

WEST

**End of Result Set**

☐ Generate Collection  Print

L26: Entry 8 of 8 File: USPT Apr 14, 1987

DOCUMENT-IDENTIFIER: US 4658351 A
TITLE: Task control means for a multi-tasking data processing system

Abstract Text (1):
A task control method and apparatus for controlling the interactive, concurrent execution of a plurality of general purpose data processing tasks in a data processing system. The system includes a memory for storing active tasks, a mass storage means for storing inactive tasks and a general purpose CPU. Upon request by a user or by an active task, a task loader transfers a presently inactive task from the mass storage means to the memory to be available for execution. A memory manager assigns a task node space in the memory and a task manager creates a task control block for the task to be activated, assigns a task control block identification to the task control block, and writes the task control block identification into the task's task node space to link the task to the task's task control block. The task manager includes a plurality of task queues, each queue corresponding to a relative priority level for execution of the tasks, and each task control block is stored in a task queue corresponding to the task's priority level. The sequence of the task control blocks in each task queue is dependent upon the status of execution of the corresponding task. Tasks are executed in a sequence depending upon the relative priorities of the task queues and upon the sequential locations of the task control blocks in a task queue.

Brief Summary Text (12):
There is a task control block for each active task in the system, and each task control block includes an identification field for storing information identifying the corresponding task, a status field for storing information identifying the status of execution of the corresponding task, a priority field for storing information identifying the relative priority of execution of the corresponding task, and a stack pointer field for storing information identifying the location of a stack mechanism usable by the corresponding task. The task control blocks reside in a task manager queue mechanism, with the task control blocks being linked by pointers in the preferred sequence of their execution.

Brief Summary Text (14):
The document file structure includes, for each document file, at least one page block, each page block comprising fields describing the logical dimensions of the page, fields describing the position of a cursor on the page, the cursor position indicating the logical position within the data contained in the page of an operation being performed on the data, a field indicating cursor type, a field describing page layout, and field containing a pointer to an area block. Each file structure includes at least one area block, each area block comprising a field for containing a pointer to a next area, fields containing information defining the type of area, fields defining the logical position of the area with the page, fields defining the margins of the area, fields defining the position of text appearing in the area, fields defining the relationship of the area to other areas of the page, and a field for a pointer to a column block. The structure further includes at least one column block, comprising a field identifying the format of the text appearing in the area, and at least one field for a pointer to a link block. The structure includes at least one line block, each line block comprising at least one field containing a string of at least one text character, a reference to attributes applying to the characters of the string, and references to external data items

associated with the line.

Brief Summary Text (16):
Each screen is described, in the screen manager control structure, by a virtual screen descriptor block and includes at least one viewport onto a portion of the data structure being operated upon by an associated task. Each virtual screen descriptor block includes a field identifying the associated screen, a field identifying the associated task, fields describing the associated screen, a field containing a pointer to a first viewport associated with the screen, a field containing an array of numbers identifying all viewports associated with the screen, and a field for containing a pointer to a next virtual screen descriptor block associated with a screen.

Brief Summary Text (17):
Each viewport is described by a viewport descriptor block including a field containing a pointer to a page containing the data to be displayed within the viewport, fields describing the logical location and dimensions of the viewport relative to the data to be displayed therein, fields describing the location of a cursor within the viewport, the position of the viewport being associated with the operation of the associated task upon the data, and a field for containing a pointer to a next viewport descriptor block of a viewport associated with the screen.

Detailed Description Text (8):
Referring to FIG. 2, a diagrammic representation of the physical residence of the Operating System (OS) 108 routines, Applications 110/Task 112 routines and Document Files 104 in the present system is shown. As described above, the controlling routines and data reside in the system memory space, which is comprised, as indicated in FIGS. 1 and 2, of Disc 120 and Memory 106.

Detailed Description Text (27):
Referring to FIG. 3, a diagrammic representation of the functional structure and operation of the system is presented. Indicated therein is Disc 120 memory space containing one or more Application/Load Units (A/LUs) 218 and one or more Document Files (DFs) 230 in disc file format. Also indicated is that portion of Memory 106 available for storing Tasks 112 and Document Files (DFs) 104 in document structure to be operated upon.

Detailed Description Text (57):
S/L 302 first generates a request to MM 306, by means of a MM 306 primitive (Getnode), for an area of Memory 106 to be assigned as the TNS 308 of the requested A/LU 218/Task 112. MM 306 will respond by assigning such a space and will identify the location of the TNS 308 to S/L 302. The TNS 308 associated with a given Task 112 is effectively that Task 112s memory space to be used as required in executing its' operations.

Detailed Description Text (59):
S/L 302 then writes the A/LU 218 into the assigned TNS 308 and the assigned TCDI 506 into a assigned location within the TNS 308. The A/LU 218 thereby becomes an executable Task 112 by being assigned an active TNS 308 and by being linked into TM 304 by its' associated TCDI 506, which is effectively an address, or pointer, to its' associated TCB 316 in TM 304.

Detailed Description Text (60):
Referring now to TM 304 as shown in FIG. 5, as indicated therein each TCB 316 includes a unique Identification 508 identifying that particular TCB 316, Status 510 information describing the state of execution of the associated Task 112, Priority 512 information identifying the relative priority of the associated Task 112, and a Task Stack Pointer 514 identifying the location of the associated Task 112s stack in TM 304's TSM 310. Each TCB 316 associated with a Task 112 effectively resides in TM 304's Task Queue 314 wherein the TCBs 316 are linked, or chained, in order of task execution by a series of head pointers and a tail pointer. The order of execution is determined by task priority and status and TM 304 maintains a separate Task Queue 314 for each level of priority.

Detailed Description Text (70):

Referring now to FIG. 6, a partial illustration of a DF 104 data structure is shown; the data structure is further illustrated in "Appendix C-Document Data Structure". The first element of the structure is a Page Block 602 containing a pointer to the first Area of the Page, Page size information, Page Cursor information describing the location of a cursor in the Page. The cursor position information, among other functions, defines the location within the Page wherein current operations, for example, the insertion of text, will be performed. Also included is information as to cursor type, Save information, information as to whether the Page displays formatting, and information regarding Page layout. The Page block also contain Path Pointer information pointing to a Path Block which in turn defines, for example, whether the Page contains graphic or free text information and areas to receive overflow text. The second element or elements of the data structure are one or more Area Blocks 604, each of which contains information pertaining to an associated Area of the Page. An Area Block 604 contains a pointer to the next Area Block 504, if there is more than one, information defining the type of Area, information identifying the locations of the top left and bottom right corners of the Area within the Page, and information regarding the right and left margins of the Area. A Text Contents pointer identifies the location of a Column Block 606, described below and identifying the location of text appearing in the Area, and a Layer Descriptor identifies the location of a Layer Block 608 described further below and identifying the location of graphics or image data appearing in the Area. Other fields of an Area Block 604 define the relationships between the edges of the Area and other Areas (ES), the path sequences to previous and next Areas of the Page (Path/Reference, Prev., Next), style information pertaining to text in the Area (Area Style), and information pertaining to positional characteristics of text appearing in the Area (VERTB1, HORTB1).

Detailed Description Text (71):
The next elements, the Column Blocks contain information pertaining to text appearing in the Area. The first information, Lines & Spacing, defines the number of lines appearing in a column of text and the vertical spacing of the lines. The Format Page field contains a pointer to a block of format information defining the format of the associated column. The following fields, Line 1, Line 2, and so on, contain pointers to Line Blocks 610, each containing text appearing in the associated column and information pertaining to that text.

Detailed Description Text (73):
Referring finally to the Layer Blocks 608, each Layer Block 608 contains a Graphics Descriptor or an Image Descriptor, each being a pointer to an associated body of graphics or image data appearing in the Area.

Detailed Description Text (76):
Referring to FIG. 7, the document file structure used by DM 318 included an Index Block 702 which includes a Text Page Index field 704 containing pointers identifying the locations of the Page Blocks 602 in the associated DF 104. As described above, a Page Block 602 of a DF 104 then contains the information necessary to locate the remaining information pertaining to that Page. The Index Block 702 further contains an Item Index field 706 identifying the locations of the Item File Blocks 708 in Item File 710.

Detailed Description Text (77):
The Item File Blocks 708 of Item File 710 may be chained together in the same manner as described above with reference to the document data structure and Item File 710 contains information pertaining to headers, footers and footnotes appearing in the document contained in the associated document structure. Index Block 702 further contains a Named Item Index Field 712 which contains pointers identifying the locations of the Named Item Blocks 714 in Named Item File 716. Named Item Blocks 714 in Named Item File 716 may again be chained together in the manner described above and essentially contain document data which is identified specifically by names, for example, forms containing blank spaces to be filled in as, for example, described in U.S. patent application Ser. No. 595,079, titled "Electronic Processing With Forms" and filed Mar. 3, 1984. Further description of the document file structure may be found in the appendix titled "Appendix D-File Management System", incorporated herein by reference.

Detailed Description Text (90):
Field 920 (Number Of Lines To Process) indicates the number of Lines in the Column currently being operated upon while Field 922 (Next Line To Process) identifies the next of those Lines. Field 924 (Lengths Of Blocks Used to Generate Offsets) identifies the lengths of the Blocks in the document. Field 926 (Pointer To Page Index For This Document) contains a pointer to the Text Page Index 704 of the Document File Structure (see FIG. 7). Field 930 (Pointer To Item Index For This Document) is similar to Field 928 but contains a pointer to Item Index 706.

Detailed Description Text (91):
Finally, Field 928 (Pointer to Bitmap Block) contains a point to the Document File Structure field indicated the used/un-used Blocks in the document while Field 934 (Pointer to List of Blocks) similarly contains a pointer to a list of the Blocks in the document.

Detailed Description Text (104):
As shown in FIG. 11, each VSDB 1102 includes a Screen Identifier (SID) field 1104 identifying the particular Virtual Screen 1006, a Virtual Screen Name (VSN) field 1106 containing a name identification of the corresponding Virtual Screen 1006, and a Task Identifier (TID) field 1108 identifying the Task 112 associated with the corresponding Virtual Screen 1006. Pointer To Next Virtual Screen (PNVS) field 1110 contains a pointer to the next VSDB 1102, thereby chaining together all currently active Virtual Screen 1006 descriptor blocks for access by SM 322.

Detailed Description Text (106):
As previously described, at least one, and possibly several viewports will be associated with each Active Screen 1006. Each such viewport associated with a Virtual Screen 1006 will have associated with it a Viewport Descriptor Block (VDB) 1120, the VDB 1120 in turn being associated with the Virtual Screen 1006's VSDB 1102. As shown in FIG. 11, the VSDB 1102 of a Virtual SCreen 1006 contains a Pointer to First Viewport field 1124, which contain a pointer to the VDB 1122 of the first viewport associated with that Virtual Screen 1006. The VSDB 1102 also contains an Array Of Allocated Viewport Numbers field (AAVN) 1126, which contains entries identifying the viewports associated with the corresponding Virtual Screen 1006.

Detailed Description Text (107):
Referring now to the structure of a VDB 1122, each VDB 1122 contains a Pointer To Next Viewport field 1128 which identifies the location of the next viewport associated with the corresponding Virtual Screen 1006, thereby chaining together the VDB 1122's associated with a particular Virtual Screen 1006 for location by SM 322. The VDB 1122 further includes a Pointer To Page (PTP) field 1130 which identifies the location of the Page upon which the associated viewport is logically positioned. The Pointer to Page field 1130 pointer points to the first Page Block 602 of that Page, as previously described with reference to FIG. 6.

Detailed Description Text (109):
The VDB also contains a Horizontal Position Of Viewport Relative To Page (HPVRP) field 1140 and a Vertical Position Of Viewport Relative to Page (VPVRP) field 1142 which describe the relative logical position of the associated viewport upon the Page identified by Pointer To Page field 1130. The information contained in the HPVRP 1140 and VPVRP 1142 fields, together with the information contained in the PTP field 1130, may be used by SM 322 to identify and locate the logical position, relative to the data contained therein, of the viewport within the DF 104 data structure being operated upon by the associated Task 112. These fields, in further conjunction with the information contained in VW field 1136 and VH field 1138, then allow SM 322 to located and identify the data logically contained within the associated viewport. SM 322 may thereby read the data so identified to R/BMDM 328 to be displayed in that area of Task Screen 1002 identified by the information contained in TLHP field 1132, TLVP field 1134, VW field 1136 and VH field 1138.

Detailed Description Text (116):
SM 322 will respond to such a request by creating the appropriate VSDB 1102 and a first VDB 1122, or if the request is for an additional viewport, by creating the appropriate VDB 1122. Considering the first, more general case, SM 322 writes a Task Identifier into TID field 1108, providing a link back to the requesting Task 112,

and a Pointer to First Viewport into PTFV field 1124 to link the VSDB 1102 to the associated VDB 1122. At this time, SM 322 writes a viewport number corresponding to the newly created viewport into AAVN field 1126. A previously described, this field is used by SM 322 in maintaining and administering the allocated viewports. Then, after creating the remaining VSDB 1102 fields, SM 322 writes a Virtual Screen Number 1202 back into a designated location in the Task 112 Task Node Space 308 to be used by the Task 112 in identifying the associated VSDB 1102.

Detailed Description Text (117):
SM 322 will then, if the request was either for a new virtual screen or for new viewport within an existing virtual screen, create a VDB 1122 as described above. If the request was for a new virtual screen, the VDB 1122 will be the first VDB 1122 and is thereby pointed to by PTFV 1124. If the request was for an additional viewport, the new VDB 1122 will chained into the already existing chain of VDB 1122's by means of a pointer written into the PTNV field 1128 of the last previously existing VDB 1122. It should be noted that, in the creation of the VDB 1122, the DCB ADD provided with the initial request is used to create the Pointer to Page field 1130 entry in the DVB 1122. The cursor type and position fields of the VDB 1122 are similarly created at this time.

Detailed Description Text (136):
Each task has a TCB (Task Control Block) associated with it. Each TCB contains a pointer field (for queuing the TCB), a task stack pointer, a unique ID, a status indicator, the task's priority, and other task-related information.

Detailed Description Text (150):
Pending task queue (FIFO queue with head and tail pointer)

Detailed Description Text (154):
Pending task pointer

Detailed Description Text (155):
Pending message queue, ordered by message priority, with head pointer on

Detailed Description Text (190):
At context switch time and at the beginning of certain primitives, 1107/OS processes the Event Mask from left to right. A bit on in the Event Mask activates the corresponding Event Handler found in the Event Table. Since these routines are activated at stable points in the 1107/OS, they can remain enabled.

Detailed Description Text (192):
Event Handlers must not call WaitSemaphore, SignalSemaphore, SetPriority, Reschedule, or any other OS primitive which could context switch. See Appendix C for further discussion of Event Handlers and ISRs.

Detailed Description Text (195):
The dynamic memory management routines should not be used excessively, since there is a certain amount of overhead. All the user tasks should allocate message nodes at initialization time and never deallocate them as possible as they can. Pointers to these nodes are kept so that they can be reused.

Detailed Description Text (198):
There are two permenant tasks on the 1107 system: the System/Loader task and the Clock Task. They are invoked at 1107 initialization, using information in the 1107 data base, and they never terminate. However, facilities are available for dynamically creating and deleting tasks. Creating a task consists of allocating and initializing its TCB, own Node Space Pool and stack, and then queuing the task to run. Any task can request termination and deletion by simply returning from its main-level procedure. 1107/OS will deallocate its TCB, own Node Space Pool and stack and delete the TCB pointer table entry.

Detailed Description Text (239):
This routine initializes the specified semaphore using the supplied count and class. The queue pointers are set to nil. Noted that all the task's semaphores are preinitialized to message semaphores. The application task only needs to

reinitialize all the semaphores used as the simple semaphores.

Detailed Description Text (257):
MessagePointer: NodePtrType=The pointer to the 7th byte of the Message Node received.

Detailed Description Text (259):
The routine operates on a message buffer semaphore belonging to the calling task. Any events specified by the Event Mask are processed first. The semaphore count is decremented and checked. A message is extracted from the semaphore's pending message queue and the operation returns immediately if the new semaphore count is greater than or equal to 0. Otherwise, the semaphore's pending task pointer is set to point to the current task and another task is scheduled. When another task or an interrupt service routine issues a Send to the message semaphore, the waiting task will be revived. The Message Pointer points to the 7th byte of the actual Node (first 4 bytes are used as a Node Link and 4th and 5th bytes are used as the size of the Node. These 6 bytes are used by the system and should not be touched by the application task).

Detailed Description Text (266):
MessagePointer: NodePtrType=The pointer to the 7th byte of the message Node to be sent

Detailed Description Text (270):
This routine operates on a message buffer semaphore. The specified message is queued on the receiving semaphore's pending message queue according to its priority (as specified in the message header). Message priorities (as opposed to task priorities) go from 255 (highest) to 1 (lowest). After queueing the message, the semaphore count is incremented and checked. Send returns immediately if the result is greater than 0. Otherwise, the receiving task is queued on the ready queue, and the routine returns to the caller. The message pointer points to the 7th by of the Message Node to be sent (first 6 byte are used by the system and should not be touched by the application task).

Detailed Description Text (300):
NodePtr: NodePtrType=The pointer to the acquired node.

Detailed Description Text (309):
NodePtr: NodePtrType=The pointer to the acquired node.

Detailed Description Text (318):
NodePtr: NodePtrType=The pointer to the acquired node.

Detailed Description Text (328):
NodePtr: NodePtrType=The pointer to the 7th byte of the node.

Detailed Description Text (350):
NameStringPtr(4 bytes): Pointer to the name string of the RUN file module to be loaded. The name string should following the legal name string convention of a RUN file defined in the FMS.

Detailed Description Text (406):
Call: GetTime(&Time); ##STR2## Output: 12 bytes ascii time returned in the input pointer. ##STR3## 3. GetDate Call: GetDate(&Date);

Detailed Description Text (408):
Output: 12 bytes ascii number of century, year, month and day were put in the input pointer.

Detailed Description Text (410):
This routine is to retrieve primary date then convert to ascii number and stored in the input pointer with the following format: XX:XX:XX:XX: formated according to the confifuratin file

Detailed Description Text (414):

Output: return 8 bytes of primary date and time to input pointer.

Detailed Description Text (440):
MsgPtr: word--Content of message pointer which the message will be sent.

Detailed Description Text (470):
NodePtr: NodePtrType=The pointer to the acquired node.

Detailed Description Text (479):
NodePtr: NodePtrType=The pointer to the acquired node.

Detailed Description Text (489):
NodePtr: NodePtrType=The pointer to the acquired node.

Detailed Description Text (499):
NodePtr: NodePtrType=The pointer to the 7th byte of the node.

Detailed Description Text (559):
When a file is opened in standard (record I/O) mode, a buffer pointer and buffer size (in bytes) must be provided by the caller. It need not be supplied, and will always be ignored, if the physical I/O option flag is set, In all cases, the user program should never access the buffer directly, since its contents are known only to FMS and/or the corresponding device driver at any given time.

Detailed Description Text (572):
Namestring Pointer--A pointer to the null-terminated filename, to be specified at Open-time.

Detailed Description Text (600):
The OPEN file access routine and all of the VTOC access routines take one or more pointers to terminal (i.e., file) or nonterminal (i.e., directory) namestrings. These namestrings are in the form of character strings that are terminated by a null (i.e., X'00') value (for "C" string usage). The text of a namestring must be as follows:

Detailed Description Text (626):
2. All direct pointer arguments in SMALL mode are automatically changed to LARGE form by the aforementioned macros, and thus need not be a user concern. However, contained pointers, such as those within the FCB, must be explicitly constructed as 4-byte values by the caller; the base portions will be filled in by FMS. Currently, there are three such pointer fields in the FCB and one each in the request array arguments to FmsCheck, FmsReadAttr, and FmsUPdateAttr. These will be noted in the documentation of the respective functions.

Detailed Description Text (629):
Note that the application is expected to set the filename pointer and open options in the FCB. Note also that ReadVtoc and ReadAttr VTOC access routines are available for obtaining the names and attributes of existing files.

Detailed Description Text (634):
fcbptr: A pointer to a properly-constructed FCB.

Detailed Description Text (648):
fcbptr: A pointer to a properly-constructed FCB.

Detailed Description Text (661):
fcbptr: A pointer to a properly-constructed FCB.

Detailed Description Text (672):
fcbptr: A pointer to a properly-constructed FCB.

Detailed Description Text (686):
fcbpter: A pointer to a properly-constructed FCB.

Detailed Description Text (697):

fcbptr: A pointer to a properly-constructed FCB.

Detailed Description Text (708):
fcbptr: A pointer to a properly-constructed FCB.

Detailed Description Text (731):
Small mode "C" uses 2-byte pointers and thus must allocate an additional "int" for the base portion of the address field. The FMS will take care of filling in the base value from register DS.

Detailed Description Text (735):
2. All direct pointer arguments in SMALL mode are automatically changed to LARGE form by the aforementioned macros, and thus need not be a user concern. However, contained pointers, such as those within the FCB, must be explicitly constructed as 4-byte values by the caller; the base portions will be filled in by FMS. Currently, there are three such pointer fields in the FCB and one each in the request array arguments to FmsCheck, FmsReadAttr, and FmsUpdateAttr. These will be noted in the documentation of the respective functions.

Detailed Description Text (744):
name: a pointer to a file or directory name.

Detailed Description Text (758):
oldname: a pointer to the old file or directory name.

Detailed Description Text (759):
newname: a pointer to the new file or directory name.

Detailed Description Text (772):
name: a pointer to the full pathname of the new directory name.

Detailed Description Text (784):
name: a pointer to the new directory name.

Detailed Description Text (800):
path: a pointer to a pathname. To read a file or directory on a disk volume, this must be set by the application to the name of the containing directory (with null equal to the root directory). This primitive will return only members of that directory, based on the input value of "element" and on the flags byte. To read volumes on the system, this field must be set to a ";" character, followed by a null.

Detailed Description Text (802):
a pointer to the element-on-path name. This argument is the "base" element name, which is used to find the "equal", "next", or "previous" element as specified by the flags byte (see below). The value of the argument is replaced by the located element upon successful return from the FmsReadVtoc call.

Detailed Description Text (808):
typptr: a pointer to a 1 byte element-type field to be filled in by ReadVTOC. Element types are 0 (for directory) and 1 (for file). The value of this argument is ignored on input. Also, it is current not used for reads to a volume.

Detailed Description Text (827):
The DCB is maintained by the Document Manager and should not be accessed by an application. A pointer to the DCB is passed as a parameter to document management service routines. This is how the calling routine specifies the document that is to be operated upon. The DM maintains sole responsibility for the contents of the DCB.

Detailed Description Text (855):
documentname: A pointer to the namestring of the document.

Detailed Description Text (856):
password: A pointer to the document password (a null terminated string). This is a a NULL pointer if no password is to be specified.

Detailed Description Text (857):
dcbptr: The location where a pointer is to be returned to the newly generated DCB.

Detailed Description Text (859):
dcbptr: Gets a pointer to a DCB block used to access the requested document.

Detailed Description Text (872):
documentname: A pointer to the namestring of the document.

Detailed Description Text (877):
password: A pointer to the document password (a null terminated string). This is a NULL pointer if no password is specified.

Detailed Description Text (878):
dcbptr: The location where a pointer is to be returned to the DCB that is generated in this routine.

Detailed Description Text (880):
dcbptr: Gets a pointer to an DCB block used to access the requested document.

Detailed Description Text (890):
dcbptr: A pointer to the address of the DCB for the document.

Detailed Description Text (902):
documentname: A pointer to the namestring of the document.

Detailed Description Text (903):
password: A pointer to the document password. This is a NULL pointer if no password exists.

Detailed Description Text (918):
dcbptr: A pointer to a DCB block used to access the document requested.

Detailed Description Text (936):
dcbptr: A pointer to a DCB block that is used to access the requested document.

Detailed Description Text (939):
itemptr: A pointer to the data item that is referenced.

Detailed Description Text (954):
dcbptrsrc: A pointer to a DCB block defining the source document.

Detailed Description Text (957):
dcbptrdest: A pointer to a DCB block defining the destination document.

Detailed Description Text (971):
dcbptr: A pointer to a DCB block that is used to access the requested document.

Detailed Description Text (988):
dcbptr: A pointer to a DCB block used to access the requested document.

Detailed Description Text (989):
pageptr: A pointer to the page structure to be written.

Detailed Description Text (1006):
dcbptrsrc: A pointer to a DCB block that defines the source document.

Detailed Description Text (1009):
dcbptrdest: A pointer to a DCB block that defines the destination document.

Detailed Description Text (1024):
dcbptr: A pointer to a DCB block used to access the requested document.

Detailed Description Text (1025):

pageno: A pointer to a location to which this routine is to return the current page for the referenced document.

Detailed Description Text (1040):
dcbptr: A pointer to a DCB block used to access the requested document.

Detailed Description Text (1042):
pageptr: The location where a pointer is to be returned to the page structure generated in this routine.

Detailed Description Text (1044):
pageptr: Gets a pointer to a page structure built by this routine containing page data to be edited.

Detailed Description Text (1054):
pageptr: A pointer to the page structure to be purged.

Detailed Description Text (1069):
dcbptr: A pointer to a DCB block used to access the requested document.

Detailed Description Text (1071):
pageptr: A pointer to the page structure to be written.

Detailed Description Text (1096):
vsno: A pointer to the location where the virtual screen number is returned.

Detailed Description Text (1102):
errorp: A pointer to the location where the error code is returned. If errorp has a value of 0, then there was no error code generated.

Detailed Description Text (1103):
name: A pointer to a SIX-character string that identifies the virtual screen. This character string is actually the icon that displays when more than one virtual screen exists (512k version only). The user can then display the virtual screen by pressing a program function key on the keyboard.

Detailed Description Text (1104):
page: A pointer to the page descriptor, to be assigned to the root viewport of the newly created virtual screen. A NULL pointer makes the screen manager leave the current screen contents unmodified.

Detailed Description Text (1110):
errorp: A pointer to the location where the error code is returned. If errorp has a value of 0, then there was no error code generated.

Detailed Description Text (1134):
page.sub.-- ptr: A pointer to the page descriptor and the data associated with the viewport.

Detailed Description Text (1166):
nvsnop: Pointer to location where the viewport number of the newly created viewport is stored (virtual screen number in the upper byte, viewport number in the lower byte).

Detailed Description Text (1167):
errorp: Pointer to location where any error code that was generated as a result of a call to this function is stored. No error code was generated if the value at this location is 0.

Detailed Description Text (1177):
display.sub.-- request.sub.-- blk.sub.-- ptr: This is a pointer to the paint request block, as discussed in section 4.4. Note that the second entry can have 5 different values. They are:

Detailed Description Text (1194):

pptr: This is a pointer to the data area where the current size of the designated viewport, and its current page window position, is stored. Note that this area is in the application task space.

Detailed Description Text (1196):
Stores the width, height, and the horizontal and vertical positions of the specified viewport on the page consecutive locations pointed to by the pointer, pptr.

Detailed Description Text (1275):
name: A pointer to a six-byte long character string that indentifies the new name of the icon.

Detailed Description Text (1276):
exception.sub.-- word: A pointer to the location where the status condition will be returned.

Detailed Description Text (1290):
exception.sub.-- word: A pointer to the location where the status condition will be returned.

Detailed Description Text (1296):
4.3.14 Getting a Page Pointer

Detailed Description Text (1297):
Use the screen manager primitive "vs.sub.-- get.sub.-- page.sub.-- ptr" to get the page pointer of a specified viewport. Note that a null pointer will be returned by this function if a page has not previously been assigned to the specified viewport.

Detailed Description Text (1304):
page.sub.-- ptr: indicates the location where the corresponding page pointer should be returned.

Detailed Description Text (1305):
exception.sub.-- word: a pointer to the location where the condition code will be returned.

Detailed Description Text (1323):
& exception.sub.-- word: a pointer to the location where the condition code will be returned.

Detailed Description Text (1335):
This is a pointer to the next request block. By using this field, the application task can issue multiple request blocks in a single update request to the screen manager. This field is null there are no other paint request blocks following it.

Detailed Description Text (1343):
If this bit is set to 1 then the screen manager will issue a request to the rasterizer to update the viewport with the specified data. If the bit is 0 the screen manager will request to the rasterizer to clear the specified area. Note that if the application task wants to clear the viewport and it specifies partial area (see Scope), the data to be cleared on the screen must be in the data pointer structure.

Detailed Description Text (1364):
4.4.4 Data Pointer

Detailed Description Text (1365):
The data pointer field points to the area that is to painted or cleared, as the case may be. This field is relevant only if the update is to be for an area or partial area (see Scope). The contents of this field are ignored if the update is to be for a virtual screen or for a viewport.

Detailed Description Text (1366):
Note the pointer in this field points to an area descriptor if text has been specified in the data type field (see Data Type). If graphics has been specified in

the data type field, this pointer points to a graphic primitive.

Detailed Description Text (1457):
The menu driver will initially call the associate menu primitive, passing the pointer to menu page descriptor and optionally a virtual screen number. The associate menu primitive will build a data structure and return a menu number to the menu driver. This menu number will then be used by the menu driver for all subsequent primitive calls to the menu peon. It is the responsibility of the menu driver to create a virtual screen for displaying the menu. If the menu driver provides the virtual screen number, the associate menu primitive will call the screen manager to display the menu page.

Detailed Description Text (1460):
The set field primitive allows the menu driver to initialize one or more fields in the menu page. This primitive will copy the data supplied for a field into the menu page structure and then call the screen manager to update the menu display. The menu driver calls this primitive with the menu number and a variable length parameter block which is composed of field numbers of the fields that need to be updated and a pointer to the data field. This primitive will copy the string attributes as well as the string value into the menu page field. It will return the number of bytes copied into the menu field. The set primitive may be called by the menu driver at any time to change the value of the menu field.

Detailed Description Text (1471):
menu.sub.-- no.sub.-- ptr--Pointer to location's where the menu number and the field number are returned.

Detailed Description Text (1472):
menu.sub.-- page.sub.-- ptr--Pointer to the menu page descriptor.

Detailed Description Text (1493):
menu.sub.-- fld.sub.-- tptr--Pointer to the table of field numbers and pointers to the initialization fields. The table consists of a count of the number of entries in the table and an entry for each field to be initialized. The field entry consists of a field number and a pointer to the field contents.

Detailed Description Text (1497):
This function causes the menu page fields to be initialized by the specified values in the menu field table. The maximum number of fields in a menu page is limited to 255. Each field in the menu page is numbered sequentially in the order of occurence in the menu page. A field number of zero will be used to iniatilize a menu page with the specified pick selections. In this case, the pointer to the field value points to a sixteen word array, each bit in the array corresponds to field. If a bit in the array is set, the corresponding field number is selected.

Detailed Description Text (1498):
For non zero field numbers, the pointer points to the location which contains the value by which the field is initialized. If the pointer is NULL, then that field is not initialized.

Detailed Description Text (1505):
menu.sub.-- fld.sub.-- tptr--Pointer to the table of field numbers and pointers to the location where field values are returned. The table consists of a count of the number of entries in the table and an entry for each field whose value is to be returned. The field entry consists of a field number and a pointer to the location where the value of a field is returned.

Detailed Description Text (1509):
This function causes the value of the specified menu page fields in the menu field table to be returned. The maximum number of fields in a menu page is limited to 255. Each field in the menu page is numbered sequentially in the order of occurence in the menu page. A field number of zero will be used to return the value of picked fields. In this case, the pointer to the field value points to a sixteen word array, each bit in the array corresponds to a field. If a bit in the array is set, the corresponding field number is selected.

Detailed Description Text (1510):
For non zero field number, the pointer points to the location where the field value is returned. If the pointer is Null, no value is returned.

Detailed Description Text (1527):
menu.sub.-- edit.sub.-- req--Pointer to menu edit request block. The request block consists of the menu number, field number, the input key code, and the error return location. The field number is updated as the cursor moves from field to field. If a valid key is encountered, the error location is zero, else it contains the error code.

Detailed Description Text (1535):
The field table list is a variable length list of field entries. Each entry consists of a field number, a pointer to the data location, the field length and a reserved word. ##STR10##

Detailed Description Text (1536):
Field numbers are assigned sequentially as they occur within a menu page from 1 to 255. The pointer points to a variable length text line. The defintion of the text line is the same as that in the word processing software architecture manual.

Detailed Description Text (1539):
Field data pointer is a pointer to the location where the value of a field is retrieved from for the set field primitive or it is the location where the value of a menu page field is returned. The field value is a text line consisting of one or more strings terminated by the null string. The field length should be large enough for the largest vield value including all headers and the terminating null string.

Detailed Description Text (1561):
crsptr1--Pointer to the cursor block of example page.

Detailed Description Text (1574):
Input: crsptr pointer to page D.S.

Detailed Description Text (1588):
crsptr--pointer to cursor block of page to be copied

Detailed Description Text (1596):
Function: This function copies the cursor block and page pointed to in the block. It allocates memory for the new page and copies format pages associated with the page to be copied. If type 0 is chosen the entire page is copied, including all the lines. If type 1 is chosen, the page is copied through the column block. However, there is only space for 1 line pointer allocated and the line pointer is NULL.

Detailed Description Text (1607):
ppge--pointer to page to be copied

Detailed Description Text (1611):
Output: ppnewpge--contains pointer to new page

Detailed Description Text (1615):
Function: This function copies the page pointed to. It allocates memory for the new page and copies format pages associated with the page to be copied. If type b 0 is chosen the entire page is copied, including all the lines. If type 1 is chosen, the page is copied through the column block. However, there is only space for 1 line pointer allocated and the line pointer is NULL.

Detailed Description Text (1625):
fmtcrsptr--Pointer to cursor block of format page D.S.

Detailed Description Text (1626):
txtcrsptr--Pointer to cursor block of text page D.S.

Detailed Description Text (1647):

Input: txtcrsptr--Pointer to cursor block of text page D.S.

Detailed Description Text (1663):
txtcrsptr--pointer to cursor block of text page D.S.

Detailed Description Text (1682):
txtcrsptr--Pointer to cursor block of text page D.S.

Detailed Description Text (1683):
txtatt--Pointer to two word string header with text attributes.

Detailed Description Text (1689):
shrt--pointer to short word wrapping flag.

Detailed Description Text (1692):
shrt--pointer to short paint flag.

Detailed Description Text (1709):
Input: txtcrsptr--Pointer to cursor block of text page D.S.

Detailed Description Text (1711):
txtatt--Pointer to text attributes.

Detailed Description Text (1712):
character--Pointer to character within the data structure.

Detailed Description Text (1714):
Function: This function outputs two pointers pointing to a character and its attributes within a line data structure. The character is specified by the cursor block's area, line, string and byte members.

Detailed Description Text (1723):
Input: txtcrsptr--Pointer to cursor block of text page D.S.

Detailed Description Text (1740):
txtcrsptr--Pointer to cursor block of text page D.S.

Detailed Description Text (1758):
Input: txtcrsptr--Pointer to cursor block of text page D.S.

Detailed Description Text (1773):
txtpgeptr--pointer to cursor block of text page D.S.

Detailed Description Text (1803):
txtpgeptr--pointer to cursor block of text page D.S.

Detailed Description Paragraph Table (2):
DATA STRUCTURES FIG. 2.1.2: PAGE DESCRIPTOR ##STR4## ##STR5## ##STR6## Page Layout HorTbl - Horizontal table parameters . . . . . . . .xxxxxxxx Number of areas on the page (default - 1) xxxx . . . . L--left, M--middle, R--right 0 . . . . . . . . . . . . . . Page with wrapped text (default) . . . .xxxx 0100: Numeric Column, left aligned 1 . . . . . . . . . . . . . . . Page w/o wrapped text Text Contents 0101: Numeric Column, right aligned .0 . . . . . . . . . . . . . . . Page with unused areas (default) 0110: Numeric Column, decimal aligned .1 . . . . . . . . . . . . . . Page w/o unused areas Pointer to Text Descriptor 0111: Numeric Column, comma aligned . .1 . . . . . . . . . . . . . . Overflowed page 0110: Numeric Column, decimal aligned . .0 . . . . . . . . . . . . . . Nonoverflowed page (default) Margins: Width (dX) . . .1 . . . . . . . . . . . . Page was edited after last pagination VerTbl - Vertical table parameters . . .0 . . . . . . . . . . . . . Page wasn't edited . . . (default) . . . .1 . . . . . . . . . . . . "Hard" (required) page break OF--Overflow H--header, T--top, M--middle, . . . .0 . . . . . . . . . . . . "Soft" page break (default) (for wrapped text areas) B--bottom, S--single, C--clmn header . . . . .0 . . . . . . . . . . Portrait (default) 10 - text overflow in the area . . . . .1 . . . . . . . . . . . Landscape 00 - text fitted in the area Area No.: in area link order . . . . . .0 . .

. . . . . . . . Paint standard size (default) 01 - free space in area . . . . . . .1 . . . . . . . . . Paint 1/2 size 11 - unknown Path number . . . . . . .0 . . . . . . . . . Te xt Page . . . . . . .1 . . . . . . . . . Format Page -1 - graphic, free text, etc 0 - default path for all nonassigned ES--Edge specifications by user wrapped text areas (no text flows between these areas) 1xxx - top edge of area can be moved up and down together with top edge of left neighbor only 1:M - Logical paths for wrapped 0xxx - top edge of area can be moved up and down independently of top edge of left neighbor text through whole document x1xx - top edge of area can be moved up and down together with top edge of right neighbor only x0xx - top edge of area can be moved up and down independently of top edge of right Prey, area & Next area - neighbor xx1x - bottom edge of area can be moved up and down together with bottom edge of left path sequence (area numbers) neighbor only xx0x - bottom edge of area can be moved up and down independently of bottom edge of left neighbor xxx1 - bottom edge of area can be moved up and down together with bottom edge of right -1.(11 . . . 1) - out of range neighbor only xxx0 - bottom edge of area can be moved up and down independently of bottom edge of 0 - start or end of a path right neighbor ##STR7##

Detailed Description Paragraph Table (3):
FIG. 3-1. File Control Block Structure Byte (s) Type Contents 1-4 addr Namestring pointer (OPEN) 5 byte File status flag byte (must be zero before OPEN): X'80' = File is open (reset at CLOSE-time) other bits reserved for use by FMS 6 byte Return code from FMS function 7 byte Error code from FMS function 8 byte Reserved (must be 0) 9-12 addr Buffer Area address (to be used internally by FMS) 13-16 long Buffer size in bytes (0 if no buffer to be used) 17-20 addr Record Area address (Read/Write) 21-24 long Requested record size in bytes (Open/Read/Write) 25-28 long Actual record size in bytes (Read/Write) 29-32 long Relative/Absolute record or byte number (used for Point/Read/Write operations) 33 byte Open mode: X'01' = read-only X'02' = write-only - low nibble X'03' = read/write X'10' = old file X'20' = new file -- high nibble X'30' = work file (not implemented) X'40' = shared file (not implemented) 34 byte File type: X'00' = sequential fixed-length (not yet implemented) X'01' = sequential delimited X'02' = stream (character) 35 byte Open Options: X'80' = AutoDelete/-AutoCreate X'40' = Physical I/O (disk only) X'20' = EOF-enable (non- disk only) 36 byte I/O Options: X'80' = NoWait option (Read/Write) X'60' = Point flags (Read/Write) (see next page for interpretation) X'08' = Priority Request flag 37 byte Delimiter flag (must be 0 for first release) 38 byte Delimiter to be used for sequential delimited files 39-40 char-2 ASCII file type (e.g., WP) 41-52 char-2 Reserved (must be 0 at Open) 53-60 char-8 Device specific status bytes 61- 128 -- Reserved (must be 0 at Open)

Detailed Description Paragraph Table (5):
typedef struct .sctn.union .sctn.char (*filptr)[ ]; int filint[2];e filunion; char filestat; /* file status flags byte */ char retcode; /* return code */ char errcode; /* error code */ char version; /* FCB version (must be 0!) */ union .sctn.char (*bufptr)[ ]; int bufint[2];e bufunion; long bufsize; /* buffer size */ union .sctn.char (*recptr)[ ]; int recint[2];e recunion; long reqrsize; /* requested record size */ long actrsize; /* actual record size */ long recnum; /* record number/offset */ char openmode; /* open mode indicators */ char rectype; /* record type indicators */ char openflag; /* open flags */ char ioflag; /* I/O flags */ char dlimflag; /* delimiter flags (must be 0) */ char dlimchar; /* delimiter character */ char fileorg[2]; /* file organization (ASCII) */ char spare1[12]; /* reserved bytes (must be 0) */ char devstat[8]; /* device-specific status bytes */ char fmsarea[96]; /* FMS-only area */ e FCB; #define filename filunion.filptr /* simplified file pointer */ #define bufarea bufunion.bufptr /* simplified buffer pointer */ #define recarea recunion.recptr /* simplified record pointer */ #define extsize bufsize /* additional extent size (bytes) */ #define fmsstat(x) ((x.retcode 8)+(x.errcode&0x00FF)) /* Word status */

Detailed Description Paragraph Table (9):
Byte(s) Contents
0 0 = FMS completion (FCB passed) 1 = Signal (signal semaphore) 2 = Message (message semaphore) *1 Task number (for

signal/message items only). (Default value of 255 equals the current task.) 2 Semaphore number (for signal/message items only) 3 Active/inactive flag, where: 0 = active check item (so process it) -1 = inactive check item (so ignore it) 4-7 A pointer for FMS and message types FMS = pointer to the FCB for the operation Message = pointer to message receiver Signal = 0 (reserved)
______________________________________ *Currently, the task number is required to be that of the current task,? either explicitly or through the default of 255.

Detailed Description Paragraph Table (10):
______________________________________ "C" Type Contents
______________________________________ int attribute number char* pointer to a new attribute value int return code (that is set by - FmsUpdateAttr)
______________________________________ Attribute ID Format Contents
______________________________________ File and Directory Attributes (all ID values are decimal) CREATOR 00 char[8] Name of the file/directory creator (user ID) FILECLASS 01 char[8] Protection class for the - file/directory (TBD) CREDATE 02 char[4] Creation date (Packed YYYYMMDD) REFDATE 03 char[4] Reference date (Packed YYYYMMDD) MODDATE 04 char[4] Modification date (Packed YYYYMMDD) BAKDATE 05 char[4] Backup date (Packed YYYYMMDD) File-only Attributes BLOKCNT 06 long Count of blocks currently used for file data EXTCNT 07 int Count of extents allocated to the file EXTSIZE 08 long Default blocks-in-extent for allocation BYTESLAST 10 int Byte count in last used block FILETYPE 11 char(2) File type (ASCII, "WP", etc.) Directory-only Attributes MEMBERS 09 int Current count of members in the directory
______________________________________

Detailed Description Paragraph Table (15):
______________________________________ Data Item Class: Item Ia: Returnd(itemget)/Put(itemput): ______________________________________ DMCDSKALOC (block #) A 2-byte block number field For "itemget", the item ID con- tains the block number to which the allocated block will be chained. "itemptr" will contain the allocated block number For "itemput", the item ID con- tains the block number to be released for reuse. DMCDSKIO (block #) A pointer to a 512-byte buffer For "itemget", the item ID con- tains the block number to be read. "itemptr" references the address of a buffer acquired by the Document Manager to receive the contents of the block referenced. (Note: It is the responsibility of the calling routine to return this buffer via ReturnMyNode.) For "itemput", the item ID con- tains the block number to be writ- ten. "itemptr" contains the address of a buffer (direct) containing the data to be written. ______________________________________

CLAIMS:

3. The task control means of claim 2, wherein each task control block further includes:

sequence pointers linking together the task control blocks residing in each task queue in the preferred sequence of execution of the tasks, and wherein

the task manager means is responsive to the conclusion of execution of a presently executing active task for reading the task control blocks of the active task residing in the task queues to determine a next active task to be executed, the task manager means being responsive to the sequence of task control blocks in each task queue and to the relative priorities of the task queues for executing the active tasks in order of priority as determined by the task queues in which they reside and in the sequence in which the task control blocks reside in the task queues.

4. The task control means of claim 2, wherein the task manager means further comprises:

stack means for storing information regarding the execution of presently active tasks,

the stack means including a task stack mechanism for each active task, and wherein

each task control block further includes a stack pointer field for storing information identifying the location of the corresponding task stack mechanism.

9. The method of claim 8, further comprising the steps of:

responsive to the relative priority of execution of active tasks and the status of each task

providing in each task control block sequence pointers linking together the task control blocks residing in each task queue in the preferred sequence of execution of the tasks, and

responsive to the sequence of task control blocks in each task queue and to the relative priorities of the task queues,

executing the active tasks in order of priority as determined by the task queues in which they reside and in the sequence in which the task control blocks reside in the task queues.

10. The method of claim 8, further comprising the steps of: p1 providing a stack means for storing information regarding the execution of presently active tasks, and

during the step of creating a task control block for a task which is to become an active task, generating a task stack mechanism for the task, so that

the stack means includes a task stack mechanism for each active task, and wherein

each task control block further includes a stack pointer field for storing information identifying the location of the corresponding task stack mechanism.

WEST

☐ Generate Collection Print

L5: Entry 5 of 14 File: USPT Feb 5, 2002

DOCUMENT-IDENTIFIER: US 6345305 B1
TITLE: Operating system having external media layer, workflow layer, internal media layer, and knowledge base for routing media events between transactions

Abstract Text (1):
A customer-interaction network operating system for managing interactions in a multimedia communication center has an external media layer for managing media contact between customers and the communication center, a workflow layer for processing customer interactions and routing events to enterprise agents and knowledge workers; and an internal media layer for managing media contact with the agents and knowledge workers. The workflow layer captures each transaction, prepares a text version of at least a portion of each non-text transaction, and extracts knowledge from the text transaction or text version of a non-text transaction to be stored in a knowledge base for later use in routing and other management functions. All transactions, text versions, and extracted knowledge is related in storage for future analysis and use.

Brief Summary Text (6):
In a CTI-enhanced call center, telephones at agent stations are connected to a central telephony switching apparatus, such as an automatic call distributor (ACD) switch or a private branch exchange (PBX). The agent stations may also be equipped with computer terminals such as personal computer/video display unit's (PC/VDU's) so that agents manning such stations may have access to stored data as well as being linked to incoming callers by telephone equipment. Such stations may be interconnected through the PC/VDUs by a local area network (LAN). One or more data or transaction servers may also be connected to the LAN that interconnects agent stations. The LAN is, in turn, typically connected to the CTI processor, which is connected to the call switching apparatus of the call center.

Brief Summary Text (8):
In recent years, advances in computer technology, telephony equipment, and infrastructure have provided many opportunities for improving telephone service in publicly-switched and private telephone intelligent networks. Similarly, development of a separate information and data network known as the Internet, together with advances in computer hardware and software have led to a new multimedia telephone system known in the art by several names. In this new systemology, telephone calls are simulated by multimedia computer equipment, and data, such as audio data, is transmitted over data networks as data packets. In this system the broad term used to describe such computer-simulated telephony is Data Network Telephony (DNT).

Brief Summary Text (9):
For purposes of nomenclature and definition, the inventors wish to distinguish clearly between what might be called conventional telephony, which is the telephone service enjoyed by nearly all citizens through local telephone companies and several long-distance telephone network providers, and what has been described herein as computer-simulated telephony or data-network telephony.

Brief Summary Text (11):
The computer-simulated, or DNT systems are familiar to those who use and understand computers and data-network systems. Perhaps the best example of DNT is telephone service provided over the Internet, which will be referred to herein as Internet Protocol Network Telephony (IPNT), by far the most extensive, but still a subset of

DNT.

Brief Summary Text (22):
In a preferred embodiment of the present invention an operating system for managing transactions between transaction partners, including customers, business partners, agents, and knowledge workers, in a multimedia communication center is provided, comprising an external media layer for managing media contact between the customers and business partners and the communication center; a workflow layer for processing transactions and routing media events between the transaction partners; an internal media layer for managing media contact with the agents and knowledge workers; and a knowledge base for storing extracted knowledge from transactions and relationships between transactions. The workflow layer captures each transaction as a multimedia file, prepares and stores a text version of at least a portion of the transaction, associates the text version with the transaction, mines the text versions for knowledge, which is stored in the knowledge base, and uses the extracted knowledge at least for routing media events between transaction partners. Recorded transactions are threaded chronologically.

Brief Summary Text (23):
In another aspect of the invention a multimedia communication center hosted by an enterprise, the center for managing transactions between transaction partners is provided, comprising a CTI-enhanced, connection-oriented switched-telephony (COST) call switching apparatus connected to an incoming trunk line and to telephones at internal agent stations; at least one data network telephony (DNT)-capable router connected by a data link to a wide area network available to the customers of the enterprise, and by internal local-area network (LAN) to personal computers having video display units (PC/VDUs) at the internal agent stations, the DNT-capable router also adapted to receive and send digital multimedia documents; and a managing server hosting an interaction operating system. The interaction operating system comprises an external media layer for managing media contact between the customers and business partners and the communication center; a workflow layer for processing transactions and routing media events between the transaction partners; an internal media layer for managing media contact with the agents and knowledge workers; and a knowledge base for storing extracted knowledge from transactions and relationships between transactions. The workflow layer captures each transaction as a multimedia file, prepares and stores a text version of at least a portion of the transaction, associates the text version with the transaction, mines the text versions for knowledge, which is stored in the knowledge base, and uses the extracted knowledge at least for routing media events between transaction partners. Again, recorded transactions and associated data are threaded chronologically.

Detailed Description Text (14):
As previously described, CINOS comprises a multi-tiered architecture. This unique architecture comprises an external media layer for interfacing with the customer or business contact, a workflow layer for making routing decisions, organizing automated responses, recording transactions, and so on, and an internal media later for interfacing and presenting interactions to an agent or knowledge worker. An innovative concept associated with CINOS involves the use of tooled process models, knowledge bases, and other object models as base instruction for it's various functions. These modular conventions may be inter-bound with each other, and are easily editable providing a customizable framework that may conform to virtually any existing business logic.

Detailed Description Text (34):
FIG. 2 is a block diagram illustrating basic layers of the network operating system according to an embodiment of the present invention. As previously described with reference to FIG. 1, CINOS comprises three basic operating layers. They are an external media layer 83, a workflow layer 85, and an internal media layer 87. External media layer 83 interfaces directly with the customers or business contacts or partners as illustrated via customers a and b, and business contact c. The bi-directional arrows beneath each of the above mentioned participants illustrate interactive participation with CINOS on the customer side.

Detailed Description Text (36):
Workflow layer 85 comprises 3 basic function categories beginning with a content

analysis category 89 wherein textual analysis, voice analysis, IVR interaction, recording and storing takes place. A next category is context resolution 91. Context resolution involves customer identification, business process binding, preparation for routing, and so on. A third category termed interaction routing 93 comprises various processes associated with the presentation of the interaction to agents, service persons, knowledge workers, business partners, customers and the like, that is, all transaction partners. Category 93 covers queuing, skill-based routing, automated treatment, workflow models, and so on.

Detailed Description Text (38):
Internal media layer 87 provides an agent with, among other options, information about the customer or contact, information about current or historical business processes, information about current interactions and their relationship to business processes, and a knowledge-base to guide the agent or knowledge worker with interaction response and workflow. An agent a, and agent b, and a knowledge worker c are shown herein interacting with the system as illustrated via bi-directional arrows. The skilled artisan will recognize these are merely examples, and there may be many more such persons, and interactions in some instances may be routed to machines for response.

CLAIMS:

1. A method for managing transactions between transaction partners, including customers, business partners, agents, and knowledge workers, in a multimedia communication center, comprising steps of;

a) managing media contact between the customers, business partners and the communication center by an external media layer;

b) processing transactions by a workflow layer, including capturing each transaction as a multimedia file and extracting knowledge from transactions;

c) managing media contact with the agents and knowledge workers by an internal media layer;

d) providing a knowledge base for storing extracted knowledge from transactions and relationships between transactions; and

e) routing media events between the transaction partners based on the knowledge and relationships of transactions in the knowledge base.

2. The method of claim 1 wherein in step b) the workflow layer comprises further steps of;

identifying customer transactions by customer and media type;

preparing and storing a text version of at least a portion of the transaction,

associating the text version with the transaction,

mining the text versions for knowledge;

storing the knowledge in the knowledge base; and

using the extracted knowledge at least for routing media events between transaction partners.